Propriétés de correction séquentielle dans un langage parallèle à mémoire partagée

Gilbert Caplain

Thèse présentée pour obtenir le titre de

Docteur de l'Ecole Nationale des Ponts et Chaussées

Spécialité

Mathématiques-Informatique

par

Gilbert CAPLAIN

Propriétés de correction séquentielle dans un langage parallèle à mémoire partagée

Soutenue le 22 septembre 1998 devant le jury composé de :

M. Nicolas BOULEAU (Président ; Directeur de Thèse)
M. René LALEMENT
M. Gilles BERNOT (Rapporteur)
M. François THOMASSET (Rapporteur)
M. François IRIGOIN

Remerciements

Le présent travail a été développé au CERMICS dans le cadre d'un projet commun avec René Lalement et Thierry Salset. Je les remercie tous deux, René plus particulièrement pour la manière dont il a assuré la direction de ce projet, Thierry plus spécialement pour avoir développé, dans le cadre de sa propre thèse, une application du résultat théorique qui est l'objet principal du présent travail.

Je remercie Bernard Lapeyre, Directeur du CERMICS, ainsi que René Lalement, de m'avoir convaincu de présenter sous forme d'une thèse un travail qui n'avait tout d'abord pas eu cette destination.

Je suis reconnaissant à Nicolas Bouleau d'avoir accepté d'être le Directeur de cette thèse. Ses avis sur l'art et la manière de présenter une thèse m'ont utilement aiguillonné au cours de mon travail de rédaction.

Je remercie Gilles Bernot et François Thomasset d'avoir accepté de consacrer du temps à l'austère tâche de rapporteur et d'avoir contribué, par leurs avis, à l'amélioration du mémoire. Je remercie également François Irigoin d'avoir bien voulu faire partie de mon jury.

Que soient également remerciés Renaud Keriven et Jacques Daniel pour la disponibilité et la patience dont ils ont fait preuve lorsque je les ai assaillis de questions informatiques ; ainsi que les secrétaires du centre, Véronique Serre (anciennement), Imane Hamade et Sylvie Petit.

Une pensée amicale, enfin, pour le CERMICS dans son ensemble, qui procure un environnement scientifique d'une grande richesse par sa diversité et prouve quotidiennement (s'il en était besoin !) qu'excellence scientifique et bonne humeur peuvent aller de pair...

Table des matières

Chapitre 1

Introduction

Les besoins de puissance et de rapidité de calcul informatique s'accroissent sans cesse. Les machines parallèles représentent une réponse à cet accroissement. Le principe général du parallélisme en informatique, consiste à faire exécuter un programme sur plusieurs processeurs simultanément, chaque processeur prenant en charge "une partie" de la tâche (dans un sens à préciser).

Les modèles de programmation parallèle

Il existe plusieurs formes de parallélisme. Nous pouvons les envisager *par l'aval*, en considérant les formes d'architecture de machines parallèles, ou bien les considérer *par l'amont*, à travers des *modèles de programmation parallèle*. Nous inspirant de la classification de Flynn [40], nous distinguons trois formes d'architecture parallèle, correspondant assez étroitement à trois modèles de programmation parallèle. Nous allons les esquisser tout d'abord, avant d'évoquer ensuite qu'il est parfois possible – et intéressant – de concevoir un programme dans un certain modèle de programmation, puis de le compiler vers une machine cible participant d'un autre modèle.

Le modèle de programmation à parallélisme de données. Dans ce modèle, des instructions s'exécutant de manière strictement séquentielle accèdent chacune à tout un ensemble de données à la fois, par exemple à tout un tableau, traitant ces données "en parallèle". C'est dans ce modèle de programmation que se place le *calcul vectoriel.* Le langage Fortran HPF [14], par exemple, se réclame de ce modèle de programmation. Ce modèle est le plus simple, car le contrôle y reste séquentiel. Cependant, la classe d'algorithmes pour laquelle il est efficace est restreinte. L'architecture de machine correspondante est dite SIMD *(Single Instruction stream – Multiple Data stream).*

Les deux autres modèles de programmation parallèle que nous allons évoquer maintenant sont les modèles à *parallélisme de contrôle* : sur les architectures qui y correspondent, plusieurs flots d'instructions peuvent s'exécuter simul-

tanément, sur plusieurs *processeurs*[1]. Ces formes d'architecture sont dites MIMD *(Multiple Instruction stream – Multiple Data stream)*.

Le modèle distribué à passage de messages correspond à *l'architecture parallèle à mémoire distribuée* MIMD–DM *(MIMD–Distributed Memory)*, dans laquelle plusieurs processeurs dotés chacun d'une mémoire locale, communiquent par l'envoi et la réception de *messages*. Ce modèle de programmation parallèle est le plus complexe à utiliser, plaçant sur le programmeur la charge de la coordination et de l'échange de données par l'envoi et la réception explicites de messages entre des unités distinctes. Ce modèle de programmation est représenté notamment par les bibliothèques PVM [16] et MPI [15]. Ce modèle est potentiellement le plus efficace mais la programmation peut s'y révéler difficile pour des applications complexes.

Le modèle asynchrone à mémoire partagée. Dans l'architecture qui y correspond, *l'architecture parallèle à mémoire partagée* MIMD–SM *(MIMD–Shared Memory)*, plusieurs processeurs s'exécutant en parallèle échangent des données avec une mémoire commune. Cette architecture est celle des multiprocesseurs à mémoire réelle partagée, machines très répandues, dotées d'un petit nombre de processeurs. Le modèle de programmation correspondant occupe, quant à sa difficulté de mise en œuvre, une position intermédiaire entre les deux autres modèles précédemment mentionnés. *C'est à ce modèle de programmation que nous allons nous intéresser dans le présent travail.*

Evoquons deux situations dans lesquelles des programmes se situant dans un certain modèle de programmation peuvent être compilés vers une machine cible participant d'un autre modèle.

Nous avons mentionné le fait que les multiprocesseurs à mémoire partagée étaient dotés d'un petit nombre de processeurs. Cependant, des travaux de recherche récents permettent d'appliquer efficacement le modèle de programmation à mémoire partagée sur des architectures distribuées à passage de messages [24], à travers de nouvelles fonctionnalités qui permettent, en quelque sorte, de compiler un programme parallèle à mémoire partagée en direction d'un réseau de stations de travail (à mémoire distribuée et passage de messages, par conséquent), ce qui permet de disposer de ressources plus étendues, sans charges supplémentaires de programmation. Il est permis de penser que cela devrait contribuer à populariser le modèle de programmation à mémoire partagée, auquel nous nous intéressons ici.

Nous avons évoqué le langage Fortran HPF, qui se place dans le modèle de programmation à parallélisme de données. Des travaux récents visent à

1. Un *processus* est l'exécution séquentielle d'un flot d'instructions. Un *processeur* est un élément de machine ayant pour fonction d'exécuter un processus. Les deux notions ne coïncident pas nécessairement : il peut parfois se révéler utile de décrire un élément de programme comme s'exécutant en plusieurs *processus*, dans des circonstances où en réalité, ces processus s'exécutent sur un même *processeur* (en temps partagé, par exemple).

développer la compilation de ce langage vers des architectures partagées ou (surtout) distribuées [6, 10]. Là encore, la préoccupation générale est de permettre de disposer de ressources plus étendues sans rendre la programmation plus complexe.

Pour plus de détails sur ces modèles de programmation parallèle, on se reportera utilement à [40, 36].

Sur les sémantiques du parallélisme

L'utilisation de la programmation parallèle pose différents problèmes. Par exemple, apparaît le problème de l'*efficacité* de la parallélisation : comment concevoir notre programme parallèle, ou le charger en machine, pour que, exploitant au mieux les possibilités de cette dernière, il fournisse ses résultats le plus vite possible ? Mais, plus fondamentalement, se pose le problème de la *correction* (au sens de : "caractère correct" – *correctness* en anglais) d'un programme parallèle : comment spécifier la *sémantique* souhaitée de ce programme, c'est-à-dire le comportement que l'on en attend quant aux calculs effectués et aux résultats obtenus ?

Lors de l'exécution d'un programme séquentiel, sa sémantique est assez bien caractérisée conceptuellement, et modélisée sans ambiguïté dans son principe. Les exécutions d'instructions individuelles s'y opèrent suivant un ordre temporel strict – chaque instruction termine son exécution avant que l'instruction suivante commence la sienne –, et dans un *environnement* donné (c'est-à-dire un état de la mémoire). Par exemple, pour les langages impératifs séquentiels, la *sémantique opérationnelle* modélise comment une exécution d'une instruction fait passer l'environnement d'un état à un autre, les transitions entre états étant régies par des règles formelles ([39], par exemple)

Le modèle à parallélisme de données a une sémantique analogue, en fin de compte, à la sémantique séquentielle : comme nous l'avons mentionné, en effet, le contrôle y reste séquentiel.

Dans les modèles à parallélisme de contrôle, au contraire, les choses apparaissent beaucoup plus compliquées. Dès lors, en effet, que plusieurs processus s'exécutent *simultanément*, interférant avec des environnements de mémoire éventuellement *distincts*, leur sémantique est beaucoup moins claire. Par exemple, il apparaît un *indéterminisme* fondamental (tenant au fait que les processus sont autorisés, a priori, à s'exécuter à des vitesses différentes) qui n'existe pas au même titre dans le cas d'un programme séquentiel.

La modélisation du comportement de programmes à parallélisme de contrôle a fait l'objet de plusieurs travaux. En 1978, Hoare a introduit le modèle des processus séquentiels communicants *("Communicating Sequential Processes", ou CSP)* [23], dans lequel les processus communiquent par envoi et réception de messages – CSP est donc plus particulièrement adapté au modèle à mémoire distribuée. Dans CSP, les messages sont *synchrones*, en ce

sens qu'un message ne peut être émis que si son destinataire est prêt à le recevoir ("rendez-vous"), par opposition au cas *asynchrone* où le processus émetteur n'a pas besoin d'attendre (en suspendant son exécution) la réception de son message – le modèle que nous allons considérer comporte un tel mode asynchrone de communication entre processus. En 1980, Milner développa un calcul algébrique permettant de décrire de manière abstraite le parallélisme et le non-déterminisme *("Calculus for Communicating Systems", ou CCS)*, développement qui déboucha ensuite sur le π-calcul (1989) ([29], par exemple). On a dit que le π-calcul est aux langages parallèles ce que le λ-calcul est aux langages fonctionnels.

Pour de plus amples développements sur ces questions, on se reportera utilement à [2].

Evoquons brièvement différentes propriétés de correction sémantique qui ont été envisagées, dans différents contextes, avant de mentionner celle à laquelle nous allons nous intéresser. Ces propriétés concernent le modèle de parallélisme par entrelacement d'actions *(interleaving semantics)*. Nous nous plaçons ici plutôt dans le modèle de programmation à mémoire partagée – dans lequel, par conséquent, des processeurs accèdent en parallèle à des mémoires – mais ce qui va être dit peut être transposé à un modèle de mémoire distribuée.

La sérialisabilité

La sérialisabilité *(serializability)* est une condition qui a été étudiée à propos des *systèmes de bases de données* [34]. On considère une base de données susceptible d'être interrogée et modifiée par plusieurs *utilisateurs*, ici assimilés à plusieurs processus, dans notre sens, accédant à cette base comme à une mémoire partagée. Chaque utilisation se décrit comme une *transaction*, composée d'une lecture de certaines données de la base, suivie d'une écriture (c'est-à-dire d'une modification) de certaines données de la base. On peut envisager un accès séquentiel à la base dans lequel les transactions se succéderaient strictement, en ce sens que la lecture composant une transaction serait immédiatement suivie de l'écriture composant cette même transaction, sans intercalation d'une opération d'une autre transaction entre les deux[2]. La sérialisabilité, c'est la prescription que le résultat de l'application d'un ensemble quelconque de transactions en parallèle, soit le même que si ces transactions avaient été appliquées selon un tel accès séquentiel (dans un ordre non imposé). Prenons un petit exemple : soient deux transactions visant chacune à incrémenter un même compteur d'une unité. Si les

2. Dans une telle exécution, les transactions seraient *atomiques* : l'atomicité (étymologiquement : indivisibilité) d'une exécution d'une instruction, c'est la spécification que les opérations élémentaires dont se compose cette exécution s'accomplissent *comme un tout*, sans entrelacement avec toute autre exécution.

deux transactions se succèdent dans le temps, sans entrelacement, le compteur se trouvera incrémenté de 2 (comportement souhaité dans l'exemple). Si en revanche les deux transactions sont entrelacées, le second utilisateur lit le compteur avant que le premier l'ait incrémenté, et l'incrément final sera seulement de 1 (comportement indésirable dans l'exemple). La sérialisabilité exige, sur cet exemple, que les deux transactions ne puissent pas être entrelacées. Dans d'autres cas, en revanche, certains entrelacements seront autorisés. La satisfaction de cette condition de sérialisabilité pourra requérir, par exemple, dans la programmation des transactions, le recours à des *sections critiques* (chap.7).

La consistance séquentielle et la linéarisabilité

La consistance séquentielle *(sequential consistency)* d'un multiprocesseur, notion développée dans [27], est la prescription que le résultat de toute exécution d'un programme soit le même que si toutes les opérations exécutées par les différents processeurs l'étaient dans un certain ordre séquentiel (non imposé) compatible avec l'ordre d'exécution des opérations de chaque processeur.

La linéarisabilité *(linearizability)*, notion développée dans [20], est une prescription plus exigeante que la consistance séquentielle, en ce sens que chaque opération exécutée sur chaque processeur se voit attribuer un *intervalle de temps* ; à l'exigence de consistance séquentielle, on ajoute alors la prescription que, dans l'ordre séquentiel "équivalent", l'instant d'exécution de chaque opération, supposé ponctuel, soit contenu dans l'intervalle de temps associé.

Illustrons ces deux propriétés à travers un petit exemple concernant des lectures et des écritures effectuées par deux processeurs `P1` et `P2` sur deux variables `a` et `b`. `L(a,1)` (resp. `E(b,0)`) signifie : lecture de la valeur 1 sur `a` (resp : écriture de la valeur 0 sur `b`). Soit une observation des opérations suivantes sur les deux processeurs :

```
P1 :       E(a,0) ; E(b,1) ; L(a,1)
P2 :       E(b,0) ; L(b,1) ; E(a,1)
```

Cette observation est compatible avec la consistance séquentielle, en ce sens qu'il existe (au moins) une exécution séquentielle de ces 6 opérations, compatible avec la sémantique des lectures et écritures, et respectant l'ordre des opérations de chaque processeur ; par exemple (en conservant trace des processeurs d'origine) :

```
[de P1 :]   E(a,0) ;         ; E(b,1) ;         ;         ; L(a,1)
[de P2 :]          ; E(b,0) ;         ; L(b,1) ; E(a,1) ;
```

Si, en revanche, une observation plus complète de ces mêmes opérations fournit des intervalles de temps `[t1,t2]` d'exécution de chaque opération, de la manière suivante :

```
P1 :        E(a,0,[0,1]) ; E(b,1,[4,5]) ; L(a,1,[6,7])
P2 :        E(b,0,[0,1]) ; L(b,1,[2,3]) ; E(a,1,[4,5])
```

cette observation ainsi complétée n'est pas compatible avec la linéarisabilité, car la lecture `L(b,1,[2,3])` y est contrainte (par l'intervalle de temps) à se produire avant l'écriture `E(b,1,[4,5])`, et devrait être une lecture de la valeur 0, non de 1.

Consistance séquentielle et linéarisabilité peuvent s'interpréter comme des propriétés que l'on peut requérir d'un mécanisme de gestion de mémoire sur une machine parallèle à mémoire partagée. Par exemple, la consistance séquentielle est assurée dès lors que les appels à la mémoire (en lecture ou en écriture), émis par chaque processeur dans son ordre d'exécution, sont servis dans l'ordre de leur arrivée à la mémoire [27]. On peut envisager, dans une démarche expérimentale, de tester la consistance séquentielle, ou la linéarisabilité, du système de mémoire d'un multiprocesseur, sur une exécution donnée d'un programme parallèle. La complexité de ce test a été étudiée par [17] : le problème général est NP-complet. Un résultat analogue de NP-complétude a été établi pour le test général de la sérialisabilité [34].

La correction séquentielle

La propriété de *correction séquentielle* que nous allons envisager est très différente des propriétés que nous venons d'évoquer. Dans ces précédentes propriétés, il était demandé qu'une exécution parallèle d'un certain programme soit analogue, dans ses effets, à une certaine exécution séquentielle, *non spécifiée a priori*, de ce même programme. La propriété que nous allons considérer est plus exigeante, au sens suivant : dans le cadre où nous nous situons, un programme parallèle est considéré comme le résultat de la parallélisation d'un programme séquentiel donné, et nous souhaitons que le résultat de toute exécution de ce programme parallèle soit identique à celui *du programme séquentiel considéré*, et non pas identique seulement à celui d'"une certaine exécution séquentielle possible". Nous y reviendrons dans un instant.

Ce point de vue que nous adoptons ici est particulièrement adapté aux applications du parallélisme dans le domaine du *calcul scientifique*.

Contexte de la recherche

Au CERMICS, nous nous sommes intéressés au modèle de parallélisme à mémoire partagée. Dans un premier temps, notre attention s'est portée sur une extension parallèle de Fortran.

Un groupe de constructeurs de super-calculateurs a créé en 1989 le *Parallel Computing Forum* (PCF) dans le but de standardiser la syntaxe et la sémantique des extensions parallèles de Fortran présentes dans leurs systèmes. Un comité de normalisation ANSI, dénommé X3H5, est issu de ce forum. Il s'est fixé pour objectif d'établir un modèle sémantique de programmation parallèle [33] indépendant de tout langage et de l'appliquer à la définition d'extensions de langages impératifs.

Dans cette mouvance, nous avons étudié un langage que nous avons appelé Fortran X3H5, obtenu en ajoutant à Fortran77 un certain nombre de constructions parallèles X3H5 [3].

Ces dernières années, le comité X3H5 semble être tombé quelque peu en sommeil, l'intérêt ayant semblé se porter plutôt vers le modèle de parallélisme à mémoire distribuée. Cependant, il semblerait qu'un regain d'intérêt se manifeste, tout dernièrement, en faveur du modèle de programmation à mémoire partagée, qui nous intéresse ici. Comme signes de ce possible regain, on peut mentionner la proposition de standard OpenMP [32], des travaux de recherche en cours [37], ou encore les travaux que nous avons déjà mentionnés sur la compilation de programmes parallèles à mémoire partagée vers des machines distribuées [24].

Dans le domaine de la programmation parallèle, une tendance importante a consisté à fournir des *paralléliseurs*, logiciels parallélisant automatiquement des programmes séquentiels. Toutefois, des concepteurs de programmes peuvent être conduits à paralléliser eux-mêmes "directement" les programmes qu'ils conçoivent. Les deux approches présentent des avantages et des inconvénients.

Le recours à la parallélisation automatique présente l'avantage de permettre à l'utilisateur de programmer "en séquentiel", selon une démarche familière, et dans un langage séquentiel bien connu de lui, par exemple Fortran. Cette démarche présente aussi des inconvénients : en raison de l'automatisme, les paralléliseurs ne peuvent transformer *chaque* programme efficacement ; la parallélisation reste assez coûteuse et de complexité difficile à connaître ; on y utilise souvent des outils génériques devant expérimenter des méthodes variées, dans une démarche non ciblée, au détriment de la vitesse de cette parallélisation.

La programmation "directement" parallèle, en revanche, peut permettre aux concepteurs d'obtenir une meilleure performance et(ou) une meilleure compréhension de la parallélisation qu'ils obtiennent. Elle fournit plus de souplesse que la parallélisation automatique.

Le sentiment qui semble prévaloir, c'est que les deux approches devront coexister. L'utilisation de langages explicitement parallèles devrait se dé-

3. Ce travail, dans ses premières phases, a été effectué dans le cadre d'un contrat MRT (contrat n^o 90 S 0961, du ministère français de la Recherche et de l'Industrie). Il a été mené à bien par René Lalement, Thierry Salset et moi-même. Se reporter à [4, 5, 36].

velopper. Pour des discussions sur ce sujet, on se reportera à [40, 9], par exemple. Les travaux sur la parallélisation automatique sont nombreux : mentionnons par exemple ici [10, 9, 6, 18]. Nous allons nous placer dans la perspective de l'utilisation de langages explicitement parallèles. Toutefois, certaines notions que nous développons ici seraient éventuellement utilisables dans le développement de paralléliseurs automatiques.

Dans le cadre de l'utilisation de langages explicitement parallèles, il peut être utile de fournir un outil de vérification de la parallélisation ainsi réalisée – plus précisément, de vérification de la propriété de correction séquentielle que nous avons précédemment introduite. L'objectif de notre travail, au CERMICS, était d'étudier la faisabilité d'un tel outil.

Nous nous sommes donc intéressés à cette propriété de correction séquentielle, pour des programmes écrits dans un langage parallèle adapté au modèle à mémoire partagée auquel nous nous intéressons ici. Cette propriété peut se décrire comme une *équivalence sémantique* entre un programme parallèle, et la *version séquentielle* de ce programme, que nous définissons. Nous souhaitons que le programme parallèle effectue les mêmes calculs, et produise donc les mêmes résultats, que sa version séquentielle, simplement plus vite que cette dernière. C'est cette propriété d'équivalence sémantique qu'il s'agirait de pouvoir vérifier.

Vérifier la correction séquentielle

L'objet principal du présent travail est de présenter et de démontrer un théorème qui énonce des conditions suffisantes pour cette propriété d'équivalence sémantique. Ce théorème s'applique à un langage parallèle beaucoup plus général que le Fortran X3H5 auquel nous nous sommes intéressés tout d'abord. Ce théorème a fait l'objet d'une application, dans un outil de vérification de programmes parallèles écrits en Fortran X3H5, dans la thèse de Thierry Salset [36].

Certaines notions de base que nous utilisons ont été introduites dans [3]. Dans le développement des programmes parallèles à mémoire partagée, la difficulté est d'éviter les *conflits mémoire*. Il y a conflit mémoire dans une exécution parallèle, lorsque deux instructions s'exécutant simultanément accèdent à la même case mémoire (à la même variable), l'une au moins écrivant sur cette variable *(conditions de Bernstein, 1966)* (lorsque les deux accès sont en lecture, leur concomitance ne crée pas de perturbation : on ne dit pas qu'il y a *conflit* dans ce cas). Les risques de conflit mémoire sont associés à des *dépendances de données*. Une dépendance de données relie deux accès à la même variable lorsque l'un au moins de ces accès est une écriture. Afin de s'assurer que le programme parallélisé fournit le même résultat que sa version séquentielle, il faut vérifier que chaque dépendance de données est

préservée, c'est-à-dire que les deux accès correspondants s'effectuent à des instants distincts, et dans le bon ordre (une exigence que l'on traduit par l'expression : *dépendance implique précédence*).

Considérons par exemple l'élément de programme ci-dessous. Dans cet élément de programme, nous supposons que le tableau $A()$ ne figure que comme indiqué. Dans le langage que nous allons définir, ceci est une *boucle parallèle* : l'instruction `pdo` (pour *"parallel do"*) signifie que les N itérations de cette boucle peuvent être exécutées en parallèle. Nous supposons ici que le nombre N d'itérations est supérieur ou égal à 4 et nous nous intéressons par exemple à l'emplacement mémoire $A(4)$. Cet emplacement est lu par l'instruction `b` à l'itération $I = 4$; mais quelle valeur contient-il à ce moment ?

```
      pdo I=1,N
        ...
b:      ...=A(I)
a:      A(I+3)=...
        ...
      endpdo
```

Si cette boucle était exécutée séquentiellement (si c'était un `do` au lieu d'un `pdo`), cette valeur serait, dans l'exemple, celle calculée par l'instruction `a` à l'itération $I = 1$. Mais, comme la présente construction permet l'exécution des itérations en parallèle, rien n'assure que l'instruction `a` à l'itération 1 s'exécute avant l'instruction `b` à l'itération 4. Elle peut s'exécuter après, auquel cas la valeur $A(4)$ lue en `b` sera la valeur qui se trouvait dans cet emplacement mémoire avant la boucle. Les deux instructions peuvent même s'exécuter en même temps, en ce sens que les opérations élémentaires composant ces instructions risquent d'être exécutées de manière entrelacée, ce qui donnerait un résultat indéterminé.

Dans le cadre où nous nous plaçons, nous voulons éviter de tels conflits mémoire. Dans l'exemple, pour redonner à cet élément de programme le déterminisme souhaité, nous ajouterons des *synchronisations* : les instructions POST et WAIT dans cet exemple (figure 1.1), dans lesquelles figure une référence à une variable de type *événement*, ici $E()$. La sémantique de ces synchronisations est la suivante : le WAIT ne peut être franchi que si un POST se référant au même événement a été exécuté auparavant. Dans notre exemple, donc, le contenu de la case $A(4)$ sera lu par `b` à l'itération $I = 4$ après franchissement du WAIT `w` à cette même itération, donc après exécution du POST `p` à l'itération $I = 1$ (synchronisation associée à l'événement $E(4)$), donc après le calcul de $A(4)$ par `a` à l'itération $I = 1$.

Dans l'exemple, le recours à une boucle parallèle avec synchronisations, plutôt qu'à une boucle séquentielle, est intéressant surtout lorsque l'exécution des instructions ici désignées par `c` et `d` est longue.

```
      do I=1,3                          do I=1,3
        ...                               ...
        post E(I)                         continue
      enddo                             enddo
      pdo I=1,N                         do I=1,N
c:      ...                               ...
w:      wait E(I)                         continue
b:      ...=A(I)                          ...=A(I)
a:      A(I+3)=...                        A(I+3)=...
p:      post E(I+3)                       continue
d:      ...                               ...
      endpdo                            enddo
```

FIG. 1.1: *Une boucle parallèle et sa version séquentielle*

La *version séquentielle* d'un programme parallèle sera le programme séquentiel obtenu en "reséquentialisant" ce programme parallèle, c'est-à-dire en remplaçant les boucles parallèles par des boucles séquentielles, et en retirant les instructions de synchronisation (ici remplacées par des CONTINUE : des instructions qui ne font rien).

Le théorème présenté ici établit que l'équivalence sémantique entre un programme parallèle et sa version séquentielle est assurée dès lors qu'un ensemble de conditions que nous spécifions sont satisfaites. Le point important, c'est que ces conditions portent sur la version séquentielle au sens suivant : supposant la préservation des dépendances à travers les relations de précédence *telles que définies selon la sémantique séquentielle*, et quelques autres hypothèses concernant la version séquentielle également, nous déduisons l'équivalence sémantique pour toute exécution possible du programme parallèle. L'exigence de se référer à la version séquentielle – l'impossibilité de se fonder sur une exécution parallèle particulière – explique en grande partie certaines subtilités de la démonstration.

Cette exigence est motivée par la considération suivante : aussi longtemps que la correction (dans notre sens) du programme parallèle n'a pas été vérifiée par un moyen quelconque, nous n'avons aucune garantie que deux exécutions différentes de notre programme fourniront les mêmes résultats. Par conséquent, l'éventuelle vérification "expérimentale" qu'une exécution parallèle est correcte, outre son coût (par hypothèse, les programmes que l'on se préoccupe de paralléliser sont généralement coûteux en temps d'exécution !), ne garantirait rien en ce qui concerne une autre exécution ultérieure du même programme : il y a un *indéterminisme inhérent* à la parallélisation que nous considérons ici. Il nous est donc interdit de nous reposer sur une démarche expérimentale, fondée sur l'observation d'une exécution parallèle.

D'où l'idée, *dans un premier temps*, de tenter une vérification *statique*, utilisant uniquement le texte du programme. Par ailleurs, ce caractère statique va nous autoriser à considérer des programmes *paramétrés*, c'est-à-dire des *classes de programmes* différant les uns des autres par les valeurs de paramètres. Par exemple, dans de nombreuses applications, un certain programme parallèle sera destiné à être exécuté un grand nombre de fois, sur des *données* différentes : ces données seront des "paramètres" dans notre sens.

A ce sujet, *dans un deuxième temps*, il sera parfois possible de vérifier statiquement que la correction d'un certain programme parallèle *ne dépend pas des valeurs des paramètres*. Dans de telles situations, qui se présenteront assez souvent en pratique, il sera envisageable de vérifier cette correction à travers l'observation d'une exécution séquentielle particulière du programme, sur un jeu de valeurs particulières des paramètres, en bénéficiant, pour ce faire, de la référence de notre théorème à la seule sémantique séquentielle. Ainsi, lorsque cette vérification *dynamique* sera concluante, le programme parallèle sera-t-il validé pour des utilisations futures sur des données différentes.

La référence à la version séquentielle dans notre théorème a une conséquence intéressante, concernant l'utilisation qui peut être faite de l'analyse de flot de données *(dataflow analysis)* dans notre démarche. L'analyse de flot de données est l'étude de ce que l'on peut inférer, à la *compilation* d'un programme séquentiel, sur la circulation des données qui s'opérera lors de l'*exécution* de ce programme. Cette étude vise par exemple à réaliser des compilateurs optimisants. Les travaux sur l'analyse de flot de données sont nombreux : mentionnons par exemple [30, 42, 31, 1, 13, 7, 11].

Adaptée à l'étude du flot de données dans un programme *séquentiel*, l'analyse de flot de données ne se transpose donc pas d'emblée à un contexte parallèle, notamment parce qu'elle fait usage de l'ordre temporel strict dans lequel s'exécute un programme séquentiel, ordre qui, par hypothèse, n'est pas préservé dans une exécution parallèle. (Il faut cependant mentionner ici que des travaux récents, par exemple [12], développent des analyses de flot de données sur des programmes concurrents.)

En revanche, dans la mesure où notre théorème permet de ramener la propriété de correction séquentielle d'un programme parallèle à des propriétés à vérifier sur la *version séquentielle* de ce programme, c'est-à-dire en fin de compte à des propriétés d'un programme séquentiel, l'analyse de flot de données peut se révéler utile pour la vérification de ces propriétés sur le texte de ce programme séquentiel. C'est ainsi que la référence de notre théorème à la seule sémantique séquentielle, nécessaire pour que nos conclusions ne reposent pas sur une hypothèse liée à une exécution parallèle particulière, nous apporte ici un autre avantage : nous permettre d'utiliser toutes les ressources de l'analyse de flot de données en vue de tenter de vérifier les hypothèses de notre théorème sur le texte de la version séquentielle.

Plan

Dans le chapitre 2, nous présentons la syntaxe du langage que nous étudions, et nous décrivons les aspects de son modèle d'exécution qui nous intéresseront. Puis nous développons la notion de correction séquentielle, qui se décrit comme une *équivalence sémantique*.

Le chapitre 3 est consacré à la définition formelle des prédicats de dépendance et de précédence.

Dans le chapitre 4, nous donnons quelques résultats préliminaires ; notamment, nous introduisons une notion de *date d'exécution*.

Le chapitre 5 est consacré à la démonstration de notre théorème d'équivalence sémantique (théorème 1).

Le chapitre 6 esquisse des considérations sur l'application de ce théorème 1. Notamment, nous y abordons l'étude de situations où l'équivalence sémantique d'un programme ne dépend pas des paramètres – le théorème 2 donne un critère simple de vérification de cette propriété. Nous esquissons l'étude de la vérification *dynamique* de correction dans une telle situation (§6.3). Puis nous étudions une propriété assez intéressante de notre équivalence sémantique : un certain caractère *incrémental* de la vérification de cette propriété.

Dans le chapitre 7, nous esquissons une extension de nos résultats des chapitres précédents, en abordant les *sections critiques*, une structure parallèle qui nous conduira à *élargir*, c'est-à-dire affaiblir, notre notion d'équivalence sémantique.

Chapitre 2

Le langage étudié

2.1 Le langage

Le langage que nous considérons est une extension d'un langage séquentiel impératif assez général par l'adjonction de quelques constructions parallèles.

En ce qui concerne les aspects séquentiels, nous avons les affectations ; des variables de types numériques (entiers, réels...), de type logique (booléens) ; les opérations arithmétiques ou logiques usuelles associées respectivement à ces types ; des références scalaires et vectorielles (tableaux – les expressions d'indices d'un tableau seront de type *entier*). Notre langage peut comporter d'autres types et structures de variables très divers, ainsi que des pointeurs (comme en C++, par exemple), mais nos résultats, toujours valables, peuvent alors éventuellement se révéler malaisément applicables, voire inexploitables. L'utilisation de pointeurs, notamment, présentera ce risque. La préoccupation que nous garderons présente à l'esprit ici, sera que les références d'entrée/sortie dans une instruction soient relativement "claires".

Notre langage comporte les instructions structurées suivantes :

– Les boucles statiques, notées :

```
DO  <indice>=<borne_inf>,<borne_sup>
    <liste_instructions>
ENDDO
```

Les bornes de boucles, expressions entières, sont évaluées à l'entrée dans la boucle, et ne sont pas réévaluées à chaque itération. (C'est en ce sens que nous parlons ici de boucles *statiques*). L'indice d'itération ne peut pas être modifié par écriture dans la boucle. Pour la commodité, nos boucles sont *normalisées*, c'est-à-dire que l'incrément de l'indice est fixé à 1 .

– Les branchements conditionnels, notés :

```
IF  <test>  THEN  <liste_instructions>
            ELSE  <liste_instructions>
ENDIF
```

– Les boucles dynamiques, notées :

```
WHILE  <test>  <liste_instructions>  ENDWHILE
```

Ces instructions structurées peuvent être emboîtées.

Nous excluons les GOTO. C'est un fait bien connu que tout algorithme séquentiel peut être exprimé sans avoir recours à des GOTO : cette exclusion ne constitue donc pas une limitation ici ; tout algorithme séquentiel peut être exprimé dans notre langage.

Notre langage comporte des appels à des sous-programmes, avec toutefois trois restrictions importantes : tout appel à un sous-programme doit se terminer ; les sorties doivent être fonction seulement des entrées *(détermination)* ; les échanges de valeurs doivent survenir seulement à l'appel (pour les entrées) et au retour (pour les sorties) – en d'autres termes, l'appel au sous-programme doit être assimilable à une instruction simple en ce qui concerne les échanges de valeurs (nous reviendrons sur ce point).

Notre langage comporte les constructions parallèles suivantes :

– Les boucles statiques parallèles, notées PDO, spécifient que les itérations peuvent s'exécuter en parallèle. Pour le reste, elles obéissent à la même syntaxe que les boucles DO.

```
PDO  <indice>=<borne_inf>,<borne_sup>
     <liste_instructions>
ENDPDO
```

– Les sections parallèles, notées :

```
PSECTIONS    SECTION  <liste_instructions>
             SECTION  <liste_instructions>
             ........
ENDPSECTIONS
```

spécifient que plusieurs sections de code peuvent s'exécuter en parallèle.

– Des synchronisations explicites : nous considérons des *synchronisations par variables événements*, à travers des paires POST/WAIT.

```
CLEAR <evenement>
POST <evenement>
WAIT <evenement>
```

Ces trois instructions CLEAR, POST et WAIT sont les seules à avoir accès aux variables de type *événement*, les deux premières en écriture, la dernière en lecture. Un POST (resp. un CLEAR) donne à l'événement auquel il fait référence la valeur *posted* (resp. la valeur *cleared*). Les variables événements sont initialisées à *cleared*. Un WAIT a un comportement particulier : il évalue l'événement auquel il fait référence ; si cet événement est *cleared*, le WAIT *attend* et recommence plus tard ; si cet événement est *posted*, le WAIT est alors franchi (alors seulement, on dit qu'il est *exécuté*), et l'exécution passe à l'instruction suivante. C'est ainsi qu'un WAIT est conduit à attendre qu'un POST ait "posté" l'évévement correspondant, comme nous le souhaitions. Notons que le CLEAR, en "réinitialisant" les événements, permet la réutilisation ultérieure des mêmes variables événements pour des synchronisations différentes.

Nous apporterons plus loin (§2.2) des compléments nécessaires sur le comportement des synchronisations.

Dans notre langage, nous introduisons des **paramètres**, sous la forme de "variables" destinées à recevoir une valeur au démarrage du programme et non modifiées par des écritures dans le courant du programme. Ainsi, dans notre étude, un *programme* représente en fait une classe de programmes qui diffèrent les uns des autres par les valeurs des paramètres. (Par exemple, dans de nombreuses applications, des dimensions de matrices seront des paramètres en ce sens. De manière différente, en présence de programmes s'exécutant sur des *données externes*, ces dernières seront ici assimilées à des paramètres.)

Une **instance de programme** est obtenue à partir d'un programme en attribuant des valeurs constantes aux paramètres.

Dans ce qui suit, les paramètres, et les indices de boucle à l'intérieur de leur boucle, ne seront pas désignés comme des "variables". Une **variable** est un *emplacement mémoire* autre qu'un emplacement assigné à un indice de boucle ou à un paramètre. Une **référence** est un élément syntaxique désignant une variable. Par exemple, dans l'affectation :

```
A = B(I)
```

où B n'est pas un tableau de paramètres, "A" et "B(I)" sont des références ; si I est un indice de boucle et que cette instruction se trouve être exécutée pour $I = 3$, alors, dans cette exécution de l'instruction, la référence "B(I)" pointe vers la variable $B(3)$. Nous dirons que cette exécution de cette instruction *fait référence* à cette variable $B(3)$, *en entrée* en l'occurrence (et à A *en sortie*).

Références dynamiques ; ordre d'indirection

Les variables sont autorisées dans les bornes de boucles DO et PDO, les expressions de test dans les IF et (évidemment) dans les WHILE, dans les expressions d'indices de tableaux *(référence dynamique de variables)*. Notons que les variables événements figurant dans les synchronisations peuvent être des tableaux, objets possibles de référence dynamique.

Cette propriété de référence dynamique nous amène à introduire la notion d'**ordre d'indirection**. Une référence est d'ordre d'indirection 0 s'il s'agit d'un scalaire ou d'un tableau dont la liste d'expressions en indice comporte seulement des constantes, des indices de boucles englobantes et des paramètres[1]. Une référence est d'ordre d'indirection $n > 0$ s'il s'agit d'un tableau dont la liste d'expressions en indice contient des références d'ordres d'indirection $n - 1$ (pour l'une au moins d'entre elles) ou moins. (Dans les programmes usuellement rencontrés, l'ordre d'indirection est rarement supérieur à 2 .)

Donnons quelques exemples. I désignant un indice de boucle englobante, P un paramètre, $A(,)$ et $B()$ des tableaux de variables, *fonc()* une fonction,

`A(I+3,fonc(P))` est une référence d'ordre 0 ;

`A(B(2),P+B(I))` est d'ordre 1 (ses deux expressions en indice étant d'ordre 0) ;

`A(B(A(3,I)),fonc(B(2)))` est d'ordre 2, ses deux expressions en indice étant d'ordres 1 et 0 respectivement.

La notion d'instance d'instruction

Pour la commodité, dans ce qui suit, les *instructions* que nous allons considérer seront seulement des instructions simples, non des instructions structurées, sauf mention contraire ; par ailleurs, nous considérerons comme *instructions* non seulement les instructions exécutables au sens habituel, mais

1. Nous supposons ici que, dans notre langage, les seuls types de variables dans lesquels une référence peut elle-même comporter des variables, sont les tableaux. Un *scalaire* est alors une variable qui n'est pas d'un type *tableau*. Il serait aisé d'étendre cette notion de référence dynamique à d'autres types dans lesquels des références peuvent elles-mêmes comporter des variables, tels par exemple des *pointeurs*.

aussi : les têtes et fins de DO, IF, WHILE et de constructions parallèles ; et les expressions de tests dans les WHILE.

La prise en considération de boucles DO, PDO et WHILE nous conduit à définir une notion d'**instance d'instruction**. Dans une première idée, une instruction d'une boucle pouvant s'exécuter plusieurs fois, chacune de ces exécutions serait définie comme une instance de cette instruction. Par exemple, dans une boucle simple s'exécutant 10 fois, chaque instruction générerait 10 instances. C'est la définition classique (voir par exemple [40]). Cette définition classique présente une difficulté pour nous : notre langage autorisant des variables dans les bornes de boucles statiques, ainsi que les boucles dynamiques (WHILE), l'ensemble des instances générées par une instruction ne serait pas toujours connu statiquement. Il nous est donc utile d'introduire une autre définition.

A chaque instruction du programme, nous associons un ensemble dénombrable d'instances d'instruction, de manière telle que deux conditions soient remplies :

- L'ensemble des instances associées à chaque instruction est défini statiquement ;
- Une instance d'instruction est exécutée au maximum une fois dans une exécution donnée du programme.

Bien évidemment, *le fait qu'une instance soit exécutée ou non* n'est *pas*, en général, défini statiquement. En somme, une instance d'instruction correspond à *une exécution a priori possible d'une instruction, pour autant qu'on puisse le prévoir statiquement.*

A toute instruction du programme, est associé un *vecteur d'indices*, éventuellement vide, dont chaque composante prend ses valeurs dans l'ensemble des entiers relatifs. Une instance d'instruction sera alors obtenue en assignant une valeur entière à chaque composante du vecteur d'indices. Ce vecteur d'indices est défini récursivement comme suit. Soit a une instruction :

- Si a n'est pas contenu dans un DO, un PDO ni un WHILE, son vecteur d'indices est vide : dans ce cas, a génère une instance d'instruction.

  Dans les autres cas, considérons la boucle la plus interne (autrement dit la plus immédiate) contenant a. Soit c la tête de cette boucle, et $\mathbf{i}$ le vecteur d'indices (éventuellement vide) de c.

- Si c est une tête de boucle DO ou PDO, le vecteur d'indices de a est obtenu par la concaténation de $\mathbf{i}$ et d'une composante j, notée $\mathbf{i}::j$. j correspond à l'indice d'itération de la boucle.

- Si c est une tête de boucle WHILE, le vecteur d'indices de a est la concaténation de $\mathbf{i}$ et d'une composante j. Cette fois, j prend des valeurs

positives, numérotant les itérations successives a priori possibles du WHILE.

De la sorte, à travers les deux dernières règles, chaque instance $c(\mathbf{i})$ (exécutée ou non) génère une infinité dénombrable d'instances $a(\mathbf{i}::j)$ mais, dans toute exécution du programme ne conduisant pas à une boucle infinie, seul un ensemble fini de ces instances est exécuté.

2.2 Le modèle d'exécution

Afin que nos résultats présentent une bonne généralité, nous ne devons pas spécifier entièrement le modèle d'exécution de notre langage, mais en préciser seulement certaines caractéristiques que nous supposerons dans la suite. Nous nous inspirons ici de la proposition de norme X3H5 [43].

Deux notions importantes vont être introduites : la notion de *processus* et la notion d'*unité de travail*.

- L'exécution du programme commence avec un *processus initial*.
- Un processus s'exécute jusqu'à ce que survienne l'une des circonstances suivantes :
  - il atteint la fin du programme (*terminaison normale – cela ne peut arriver qu'au processus initial*) ;
  - il rencontre une construction parallèle ;
  - il rencontre la fin d'une construction parallèle ;
  - il rencontre un WAIT ;
  - il rencontre une erreur d'exécution.
- Quand un processus rencontre une construction parallèle, il devient le *processus de base* pour cette construction. Cette construction parallèle spécifie un certain nombre d'*unités de travail* : chaque itération d'un PDO et chaque section d'un PSECTIONS est une unité de travail. Une *équipe* de processus est créée. Chaque unité de travail est assignée à un certain processus de cette équipe, dans un certain ordre. Ainsi, à partir de ce point, chaque processus a en charge une ou plusieurs unité(s) de travail[2]. (Comme nous autorisons les constructions parallèles emboîtées, cette définition des unités de travail et des processus opère

2. On peut concevoir, en une variante, que cette assignation d'unités de travail aux processus soit *dynamique* : par exemple, les unités de travail générées par la construction parallèle seraient "conservées quelque part" puis affectées aux processus de l'équipe à mesure que ceux-ci se libèrent. Une telle variante peut accroître la rapidité de l'exécution, mais ne change pas fondamentalement les considérations qui suivent.

récursivement : une unité de travail peut spécifier des sous-unités ; un membre de l'équipe de processus peut à son tour devenir processus de base, et ainsi de suite.)

Lorsque le processus de base crée une équipe de processus, des copies des variables intervenant dans la construction parallèle sont constituées pour chaque processus de l'équipe ; les calculs sont alors effectués localement dans chaque processus de l'équipe.

– Lorsqu'un processus a terminé l'exécution d'une unité de travail, l'exécution passe à l'unité suivante qui lui est assignée, le cas échéant (nous dirons que l'unité suivante est *chargée*) ; si ce processus a terminé l'exécution de toutes les unités de travail qui lui avaient été assignées, il attend que les autres processus de la même équipe terminent leur travail.

– Si et quand tous les processus de l'équipe ont terminé leur travail, cela signifie que toutes les unités de travail de la construction parallèle ont été exécutées. Alors, les processus de l'équipe communiquent au processus de base les valeurs des variables qu'ils ont modifiées (on parlera de *mise à jour* des variables) ; ensuite, l'équipe est dissoute et son processus de base poursuit l'exécution. (C'est seulement alors, que nous dirons que le ENDPDO ou ENDPSECTIONS est *exécuté*.)[3]

– Une telle mise à jour de variables intervient aussi lorsqu'un processus de l'équipe exécute une instance d'un POST ou d'un WAIT. Cette mise à jour correspond au fait qu'une instance de WAIT a pour destination d'attendre l'exécution d'une certaine instance d'un POST, l'intention sous-jacente étant que – par exemple – une certaine valeur calculée avant le POST dans son processus, soit effectivement disponible juste après le WAIT dans le sien.

Nous avons expliqué (§2.1) comment un WAIT "attend" qu'un POST ait "posté" l'événement correspondant, et s'exécute seulement alors. Plus précisément, lorsque le processus parvient à une instance d'un WAIT, il examine quel événement est référencé (ce qui peut nécessiter des calculs en cas de référence dynamique). Si cet événement n'est pas "posté" à ce moment, cet examen est réitéré jusqu'à ce que, éventuellement,

3. On pourrait introduire dans notre langage la notion de *variable privée* (évoquée dans la proposition de norme X3H5 [43]), variable temporaire utilisée localement à chaque processus de l'équipe, non destinée à être communiquée entre processus, ne faisant donc pas l'objet des mises à jour évoquées ici, ni par conséquent des vérifications ultérieures de préservations de dépendances. Une telle introduction, assez simple dès lors que références "privées" et références "partagées" (nos "variables") seraient clairement distinguables, n'a pas été faite ici pour ne pas alourdir encore l'exposé. *Toutefois*, les indices de boucles DO et PDO sont supposés ici avoir ce statut de variables privées, dans d'éventuelles constructions parallèles englobantes.

l'événement ainsi examiné soit "posté" [4].

- En ce qui concerne les mises à jour de variables que nous avons mentionnées, il peut naturellement se présenter des conflits, reflétant un *indéterminisme inhérent* à l'exécution parallèle. Il n'est pas fait d'hypothèse ici sur ce qu'il advient lorsque de tels conflits se présentent. L'objet de notre démarche sera précisément de détecter si de tels conflits risquent de se présenter (circonstance indésirable) ou si nous pouvons garantir qu'il n'y en aura pas (propriété souhaitée).

  Par ailleurs, il n'est pas fait d'hypothèse sur la survenance ou non de mises à jour de variables dans des circonstances autres que celles que nous avons mentionnées ci-dessus : la terminaison d'une construction parallèle et les synchronisations POST/WAIT [5]. *Toutefois*, en ce qui concerne l'exécution d'*une* instance d'instruction par un processus, nous ferons l'hypothèse que cette instance collecte ses entrées le cas échéant, *puis* effectue ses calculs le cas échéant, *puis* produit ses sorties le cas échéant, sans interférence d'une mise à jour de variables *pendant* les calculs en question.

- Nous aborderons plus loin le problème des erreurs d'exécution.

Il est important de bien faire la distinction entre ces deux notions, très différentes, de *processus* et d'*unité de travail*. La génération des processus, au cours de l'exécution, est tributaire des ressources disponibles sur la machine à ce moment-là : c'est un phénomène *très dépendant de la machine*. Au contraire, la caractérisation des unités de travail dépend uniquement de la sémantique du programme considéré, telle qu'elle se déploie au cours de l'exécution ; dans la mesure où le programme sera correct dans notre sens, cette caractérisation sera *indépendante de la machine*.

Dans de nombreuses implémentations, le nombre maximum de processus tournant simultanément n'excède pas quelques unités. Par conséquent, il sera tout à fait courant que plusieurs unités de travail se trouvent assignées à un même processus, qui les exécutera successivement. Par ailleurs, dans le cadre où nous nous plaçons, nous souhaiterons que la correction de nos

4. Dans le modèle d'exécution d'une instance de WAIT comportant une référence dynamique, un point n'est pas précisé ici : les variables intervenant dans la référence d'événement sont-elles réévaluées à chaque tentative, ou bien sont-elles lues une fois pour toutes lorsque l'instance est atteinte (auquel cas le WAIT attend toujours le même événement) ? Nous n'avons pas besoin d'opter entre ces deux modèles d'exécution possibles : nos résultats s'appliquent dans l'une et l'autre options.

5. Remarquons cependant qu'un modèle d'exécution efficace économisera au maximum ces mises à jour de variables, car il est bien connu que les transferts de données entre mémoires sont coûteux *(von Neumann bottleneck – [40], par exemple)*. Par ailleurs, dans cette perspective d'économie, il est possible, pour limiter les mises à jour associées aux synchronisations, d'envisager des *synchronisations* POST/WAIT *à passage de références* : se reporter au §7.1.

```
a0:     A(0)=...
p0:     post E(0)
        pdo I=1,N
          ...
w:        wait E(I-1)
b:        ...=A(I-1)
a:        A(I)=...
p:        post E(I)
          ...
        endpdo
```

FIG. 2.1: *Un exemple d'utilisation du post/wait*

programmes parallèles soit assurée indépendamment du nombre de processus qui se révéleront disponibles pour une exécution. Elle devra être assurée même dans le cas limite où notre programme parallèle devrait s'exécuter *sur un seul processus* – à plus forte raison, dans le cas où une certaine construction parallèle de notre programme trouverait un seul processus disponible pour s'exécuter, au cours d'une exécution "multiprocessus" de l'ensemble du programme. Il faudra notamment éviter le phénomène de *blocage en attente* – nous y reviendrons dans un instant.

Un exemple typique de synchronisation par événements est montré dans la figure 2.1. Dans cette boucle parallèle, les synchronisations sont conçues pour s'assurer que la valeur de $A(I-1)$ utilisée par l'instruction **b** à l'itération I est la valeur calculée par l'instruction **a** à l'itération $I-1$ (ou, pour $I=1$, par l'instruction **a0**) : effectivement, l'instruction d'attente **w** à l'itération I attend que l'événement $E(I-1)$ ait reçu la valeur *posted* par l'instruction POST **p** à l' itération $I-1$ (ou par l'instruction **p0** pour $I=1$). Bien entendu, il ne faut pas que les événements E(I) se soient trouvés *posted* avant cette boucle sans devenir *cleared* entretemps, ni qu'ils soient *posted* ailleurs en parallèle, dans le cas où l'élément de programme montré ici serait lui-même inclus dans une construction parallèle englobante.

Une unité de travail en train d'attendre son exécution sur un processus sera dite *en suspens*. Plus précisément, en ce qui concerne les instances d'instructions, la première instance d'une telle unité de travail sera dite *en suspens* à ce moment-là (nous verrons plus loin (§4.2) pourquoi seule la *première* sera ainsi qualifiée)[6].

6. Dans le cas où cette instance est un WAIT, nous dirons qu'elle est *en suspens* aussi longtemps qu'elle n'est pas *chargée* ; elle devient *atteinte* aussitôt qu'elle est chargée, et *en attente* si son événement n'est pas encore *posted*.

Les erreurs d'exécution

Une *erreur d'exécution* correspond à une circonstance où l'exécution d'une instance d'instruction crée une opération *interdite* par le langage ou l'environnement d'exécution, de sorte que l'exécution s'arrête à ce point (on parlera familièrement de *plantage*). Il peut s'agir, par exemple, d'une division par zéro, ou d'un débordement de tableau, ou de l'entrée/sortie d'un objet de type non conforme à celui de la référence...

Nous devons envisager, dans notre modèle d'exécution, l'éventualité d'erreurs d'exécution. En effet, si nous ferons l'hypothèse que la version séquentielle de notre programme ne présentera pas de telles erreurs, nous ne pouvons pas supposer, *a priori*, la même hypothèse en ce qui concerne notre programme parallèle, car nous voudrons montrer des propriétés d'équivalence sémantique en nous référant seulement à la sémantique séquentielle (nous y reviendrons plus loin : §2.3).

Concernant l'effet d'une erreur d'exécution dans un processus, nous allons faire l'hypothèse simplificatrice suivante : la survenue d'une erreur à l'exécution d'une instance d'instruction dépend exclusivement des entrées sur lesquelles elle entreprend de s'exécuter et des calculs qu'elle tente d'effectuer sur ces entrées. Elle ne dépend donc pas de ce qui peut advenir simultanément sur d'autres processus tournant en parallèle (sauf, bien entendu, à travers l'effet éventuel de ces autres processus sur les entrées de cette instance). Par conséquent, lorsque le processus dans lequel survient cette erreur fait partie d'une équipe de processus (en train d'exécuter une construction parallèle), ce processus se bloque, naturellement (on parlera de *blocage en erreur*) ; mais cela ne provoque pas le blocage immédiat des autres processus en parallèle. Cette hypothèse n'est pas restrictive dans notre démarche, dans la mesure où nous démontrerons que, sous des conditions données, *il n'y aura pas* d'erreur d'exécution se conformant à cette hypothèse dans notre programme parallèle, donc pas davantage d'erreur "induite en parallèle" si nous levions cette hypothèse.

Blocages et boucles infinies

Dans le langage que nous considérons, il y a trois manières pour un programme de ne pas se terminer normalement : il peut arriver à un *blocage en erreur*, ou bien à un *blocage en attente*, ou bien entrer dans une *boucle infinie*.

Un blocage en attente implique nécessairement une instance d'instruction WAIT dont l'événement persiste à ne pas devenir *posted*. Dans le cas (habituel) où ce WAIT est localisé dans une construction parallèle, une situation de blocage en attente peut être décrite comme suit :

– Une ou plusieurs instance(s) d'instruction(s) WAIT sont atteintes mais

non exécutées ; par conséquent, les unités de travail correspondantes ne se terminent pas ;

- En conséquence, l'exécution de la construction parallèle ne peut pas se terminer ;
- En conséquence éventuelle, il peut se faire que des unités de travail ne commencent pas leur exécution parce qu'elles sont assignées à un processus après une unité de travail bloquée, alors qu'elles seraient "exécutables en principe". La première instance d'une telle unité de travail sera dite *bloquée en suspens*.
- En présence de constructions parallèles emboîtées, un blocage dans une construction englobée entraîne une situation de blocage similaire dans la construction englobante.

Une situation de blocage en erreur aura des effets très semblables à ceux d'un blocage en attente. Une petite différence à remarquer ici : tandis qu'une instance d'instruction bloquée en attente ou en suspens sera dite "ne pas être exécutée", une instance donnant lieu à un blocage en erreur "est exécutée" : c'est précisément son exécution qui produit une erreur.

Une boucle infinie peut survenir seulement du fait d'un WHILE (se souvenir ici de l'absence de GOTO et du fait que les bornes de boucles DO et PDO sont évaluées une fois à l'entrée de la boucle). L'instance ENDWHILE correspondante n'est alors jamais exécutée ; nous dirons qu'elle est *bloquée en suspens*. (Ainsi, il existe deux manières différentes, pour une instance, d'être *bloquée en suspens* ; en dépit de leur différence de nature, nous remarquerons quelques similitudes entre les deux situations.)

Lorsqu'un WHILE est inclus dans une construction parallèle, une boucle infinie dans ce WHILE entraîne une situation similaire à celle créée par un blocage.

L'assignation séquentielle

Considérons une construction parallèle, et une paire POST/WAIT associant deux unités de travail de cette construction. Dans de nombreuses situations, nous l'avons mentionné, il y aura moins de processus disponibles pour l'exécution de cette construction que d'unités de travail générées par elle. Il arrivera donc très souvent que plusieurs unités de travail soient affectées à un même processus. Si donc nous n'imposions aucune condition d'assignation des unités de travail aux processus, il pourrait advenir la circonstance suivante : l'unité de travail contenant le POST se trouverait affectée au *même* processus que celle contenant le WAIT correspondant, et *après* cette dernière, d'où une situation de blocage (le WAIT, attendant vainement

l'exécution du POST situé après lui sur le même processus, empêcherait par là-même celle-ci !). Il est donc impératif d'incorporer, dans le modèle d'exécution, une disposition évitant de tels "faux blocages". Cette disposition doit satisfaire les deux conditions suivantes :

- *Simplicité :* Cette disposition doit, bien entendu, être applicable préalablement à l'exécution effective des unités de travail en question, ne requérir aucune analyse fine des synchronisations en présence, ni, bien entendu, aucune intervention spécifique du programmeur.

- *Indépendance par rapport aux ressources disponibles :* Elle doit permettre d'éviter les situations de "faux blocage" quel que soit le nombre de processus disponibles – donc, même dans le cas limite où le programme s'exécute sur un seul processus.

Nous sommes ainsi amené à prescrire la condition suivante[7] :

Condition SA (Assignation séquentielle) *Dans toutes les constructions parallèles contenant des instructions de synchronisation (c'est-à-dire des instructions* POST*,* WAIT *et* CLEAR*), l'attribution des unités de travail aux processus disponibles s'opère de telle sorte que les unités de travail affectées à chaque processus s'y succèdent dans l'ordre séquentiel : l'ordre défini par l'indice d'itération pour les* PDO*, l'ordre textuel des sections pour les* PSECTIONS.

Cette condition d'assignation séquentielle se combine nécessairement avec une prescription de programmation selon laquelle, dans une paire POST/WAIT, l'instance du POST doit précéder l'instance du WAIT dans l'ordre séquentiel, afin d'éviter des situations de blocage même lorsque la construction parallèle s'exécute sur un seul processus disponible (situation dans laquelle la seule assignation vérifiant la condition SA correspond à l'ordre séquentiel).

La figure 2.2 présente, à titre d'exemple, des assignations de 9 unités de travail (numérotées dans l'ordre séquentiel) à 3 processus. Seule celle figurant à droite est interdite, en raison des unités de travail marquées par $*$, qui ne se trouvent pas dans l'ordre séquentiel sur leur processus respectif[8].

7. Cette condition correspond à l'utilisation de l'option ORDERED dans la proposition de norme X3H5 [43].

8. La condition SA adoptée ici est en quelque sorte une *version minimale* juste nécessaire pour obtenir nos propriétés sémantiques. Dans une implémentation de notre langage, il est permis – et recommandé – de la renforcer éventuellement par des critères d'*efficacité*. Par exemple, sur la figure 2.2 , en l'absence d'information a priori sur les synchronisations, la première distribution à gauche est plus recommandée que la deuxième à gauche, qui apparaît "presque séquentielle"...

```
P1 P2 P3        P1 P2 P3        P1 P2 P3        P1 P2 P3
--------        --------        --------        --------
 1  2  3         1  5  8         1  2  3         1  2  4*
 4  5  6         2  6  9         6  5  4         6* 7  3*
 7  8  9         3  7            9  7  8         5* 8  9
                 4
   OUI             OUI             OUI             NON
```

FIG. 2.2: *Assignation séquentielle : 3 distributions permises, 1 défendue*

Introduire des instructions de rendez-vous ?

Le mécanisme de *rendez-vous* est une forme de synchronisation que nous n'avons pas envisagée dans notre langage – nous allons voir pourquoi. Dans ce mécanisme, deux instances d'instructions appariées s'attendent l'une l'autre, en ce sens que c'est seulement lorsque l'exécution a atteint l'une et l'autre instances, chacune dans son unité de travail respective, qu'elle peut se poursuivre dans les deux processus concernés (nous avons déjà rencontré, à propos de CSP, ce genre de communication *synchrone* (chap.1)). Il est aisé de voir qu'un rendez-vous est équivalent à un système de deux paires POST/WAIT entrecroisées. Sur la figure 2.3, est représenté à gauche un exemple d'utilisation du mécanisme de rendez-vous. Le comportement obtenu est sémantiquement équivalent à celui produit par la portion de programme représentée à droite.

Il est important d'observer ceci : toutes les fois que deux unités de travail liées l'une à l'autre par un rendez-vous, se trouveront affectées au même processus, cela créera une situation de blocage. C'est pourquoi, dans le langage que nous considérons, compte tenu de notre modèle d'exécution et de la condition d'assignation séquentielle, nous ne pouvons pas introduire de structure de rendez-vous.

2.3 La correction séquentielle. La notion d'équivalence sémantique.

Par définition, la *version séquentielle* d'un programme parallèle est le résultat de la transformation de PDO en DO, l'effacement des instructions PSECTIONS, SECTION et ENDPSECTIONS, et la *désactivation* des instructions POST, WAIT et CLEAR : par "désactivation", nous voulons dire que, dans la version séquentielle définie ici, elles sont converties en instructions qui ne font rien (notées CONTINUE ici), mais nous conservons la possibilité, dans les développements qui suivent, de garder trace des références d'événements, nous autorisant à considérer ce qu'il advient de ces références, comme si elles étaient effectivement traitées par la version séquentielle.

```
psections                          psections
  section                            section
    ...                                ...
    rendezvous(E)                      post E1
                                       wait E2
    ...                                ...
  section                            section
    ...                                ...
    rendezvous(E)                      post E2
                                       wait E1
    ...                                ...
endpsections                       endpsections
```

FIG. 2.3: *Le mécanisme de rendez-vous*

Dans la plus grande partie de cette étude, nous nous plaçons dans la situation où le comportement observable (les calculs effectués sur les variables) que nous souhaitons obtenir de notre programme parallèle est celui de sa version séquentielle, qui sera supposée se conformer aux règles de notre langage, et dont la sémantique est donc bien définie. Dans ce cadre, un programme parallèle correct peut être vu comme une certaine parallélisation d'un programme séquentiel sous-jacent, et non comme un programme "fondamentalement" parallèle. L'avantage recherché à travers la parallélisation consiste alors "seulement" en la possibilité d'exécuter le programme plus vite.

La figure 2.4 présente l'élément de programme parallèle montré dans la figure 2.1, ainsi que sa version séquentielle, qui fixe la sémantique de référence : la valeur calculée par l'instance d'instruction $a(I-1)$ est utilisée comme entrée de l'instance d'instruction $b(I)$, comme spécifié par l'ordre séquentiel d'exécution de la boucle DO. Les instructions de synchronisation **w** et **p** ont été introduites dans la version parallèle afin de préserver cet ordre d'exécution lorsque le DO est parallélisé en PDO.

Notre objectif est de prouver la correction (ou la non correction) d'un programme parallèle, dans ce sens. Nous voudrions montrer que toutes les variables venant à être calculées doivent, dans les deux versions, faire l'objet des mêmes calculs et, par conséquent, présenter les mêmes valeurs (*équivalence sémantique*).

Nous imposons, dans notre langage, une condition de *détermination*, comme un prérequis pour la correction de notre programme (plus précisément, de sa version séquentielle) :

Condition de détermination *Dans la version séquentielle d'un programme, toute variable utilisée comme entrée dans une instance d'instruction exé-*

```
a0:     A(0)=...                          A(0)=...
p0:     post E(0)                         continue
        pdo I=1,N                         do I=1,N
          ...                               ...
w:        wait E(I-1)                       continue
b:        ...=A(I-1)                        ...=A(I-1)
a:        A(I)=...                          A(I)=...
p:        post E(I)                         continue
          ...                               ...
        endpdo                            enddo
```

FIG. 2.4: *Un élément de programme parallèle (à gauche) et sa version séquentielle (à droite)*

cutée a été initialisée au préalable ; de même, tout indice de boucle DO *ou* PDO *utilisé comme variable d'entrée après la boucle, a été initialisé après la boucle et antérieurement à cette utilisation.*

Sous cette condition de détermination, nous exprimons l'équivalence sémantique de la manière suivante : *toute instance d'instruction exécutée dans une quelconque exécution parallèle est aussi exécutée dans la version séquentielle, et inversement ; toute référence figurant dans cette instance d'instruction en entrée désigne la même variable, et cette variable a été calculée par la même autre instance d'instruction, dans une quelconque exécution parallèle, que dans la version séquentielle.* (En conséquence, cette variable recevra effectivement la même valeur dans les deux exécutions.)

La vérification de l'équivalence sémantique suppose de vérifier que le programme parallèle ne se bloquera pas, quel que soit le nombre de processus disponibles ; et aussi que seront évités des "conflits mémoire" (chap.1). Toutes les fois que deux instances d'instructions font référence à la même case mémoire dans la version séquentielle, l'une au moins en écriture, nous dirons qu'elles sont dans une relation de *dépendance* (notée Dep). Il nous faudra alors vérifier que ces instances d'instructions sont dans une relation de *précédence*, c'est-à-dire que la structure de contrôle du programme parallèle et les synchronisations préservent l'ordre dans lequel ces instances seront exécutées, et assurent la mise à jour nécessaire des variables entretemps. Nous retrouvons là le schéma bien connu *"dépendance implique précédence"* [3]. Insistons bien ici sur l'importance de cette notion de mise à jour de variables : le mot "précédence" risque d'induire en erreur, si l'on n'y prend pas garde, en laissant croire qu'il s'agirait *seulement* d'assurer une succession temporelle.

Par exemple, selon notre modèle d'exécution, une synchronisation POST/WAIT est une précédence dans notre sens.

L'existence d'une sémantique séquentielle de référence est une propriété

assez particulière qui n'est pas nécessairement présente dans toute étude concernant le parallélisme. De fait, nous aborderons au chapitre 7 des extensions possibles : des situations dans lesquelles l'exigence d'équivalence sémantique sera affaiblie *(sections critiques)*.

Chapitre 3

Dépendances et précédences

Le théorème que nous présentons ici se réfère à la sémantique de la version séquentielle uniquement. Le point essentiel de ce résultat est de montrer que la vérification de la relation "dépendance implique précédence" *sous la sémantique séquentielle* (valeurs "séquentielles" des variables, etc.) assure effectivement l'équivalence sémantique entre le programme parallèle et sa version séquentielle. Dans le cadre de ce résultat, nous serons conduit à considérer la condition pour qu'une instance d'instruction soit exécutée *dans la version séquentielle* (prédicat $\mathrm{Exe}^{\mathsf{s}}$), qui est bien définie dans notre langage ; le prédicat de dépendance Dep qui concerne la version séquentielle par définition ; et le prédicat de précédence $\mathrm{Pre}^{\mathsf{s}}$ qui exprime *les relations de précédence qui s'appliqueraient dans le programme parallèle, en conséquence du modèle d'exécution, si l'on suppose que les variables figurant dans la définition de ces relations reçoivent leurs valeurs "séquentielles".* [1]

Dans la suite de nos développements (chap.4 et 5), nous serons amené à considérer, avec les précautions nécessaires, les analogues parallèles de $\mathrm{Exe}^{\mathsf{s}}$ et $\mathrm{Pre}^{\mathsf{s}}$ – Exe et Pre respectivement – qui, aussi longtemps que l'on n'a pas prouvé l'équivalence sémantique, ne sont définis qu'en référence à une exécution donnée du programme parallèle (ils peuvent différer pour des exécutions différentes du *même* programme parallèle). Considérant une exécution particulière, le prédicat de précédence Pre associé à cette exécution exprime *les relations de précédence qui s'appliquent à cette exécution, en vertu du modèle d'exécution, en considérant les valeurs* dans cette exécution *des variables figurant dans la définition de ces relations.* Nous y reviendrons plus loin (§3.3.3).

1. Une conséquence de cette définition de $\mathrm{Pre}^{\mathsf{s}}$, est que la relation exprimée par $\mathrm{Pre}^{\mathsf{s}}$ s'applique en particulier à la version séquentielle elle-même. Autrement dit, pour toutes instances d'instructions α et β exécutées dans la version séquentielle, $\mathrm{Pre}^{\mathsf{s}}(\alpha, \beta)$ implique que α est exécuté avant β dans la version séquentielle.

3.1 Le prédicat d'exécution séquentielle

Soit Exe^s la condition d'exécution d'une instance d'instruction dans la version séquentielle, *sous la condition supplémentaire que le programme séquentiel se termine*, c'est-à-dire ne boucle pas sur un WHILE. Alors, Exe^s est bien défini (bien qu'il ne soit en général pas calculable !) et son expression est assez simple dans le cas de notre langage.

Pour une instruction a et un vecteur d'indices $\mathbf{i}$ tel que l'instance $a(\mathbf{i})$ soit exécutée dans la version séquentielle, nous pouvons considérer l'environnement dans lequel l'exécution de $a(\mathbf{i})$ a lieu. Pour toute expression *exp* venant à être évaluée au cours de l'exécution de $a(\mathbf{i})$, sa valeur est définie dans cet environnement : elle sera notée $[\![exp]\!]@a(\mathbf{i})$. (Remarquons que, tout appel à un sous-programme se terminant dans notre langage, l'évaluation d'une expression se termine toujours.) Remarquons bien que $[\![exp]\!]@a(\mathbf{i})$ n'est *pas défini* dans le cas où $a(\mathbf{i})$ n'est *pas* exécuté. Cela nous conduit, dans les expressions qui suivent, à faire usage de la *conjonction séquentielle*, notée &, qui diffère de la *conjonction logique*, notée $\wedge$, de la manière suivante : si A et B sont des expressions booléennes (prenant les valeurs *vrai*, *faux* ou *indéfini*), A & B est faux dès lors que A est faux, même si B est indéfini, tandis que $A \wedge B$ est indéfini dans ce cas[2].

Exprimons $\text{Exe}^s(a(\mathbf{i}))$ pour une instruction a indexée par $\mathbf{i}$. Plusieurs cas doivent être considérés, en fonction de l'inclusion de a dans une instruction structurée (boucle ou IF). Dans le cas d'une telle inclusion, nous considérerons l'instruction structurée la plus "interne" dans laquelle se trouve a, celle dans laquelle a est *le plus immédiatement contenu*. (Ici et dans tout ce qui suit, $\neg$ désigne la négation logique.)

- a n'est pas contenu dans un DO, un PDO, un IF ni un WHILE : alors, $\text{Exe}^s(a) = \mathit{vrai}$.

- a est immédiatement contenu dans un IF de tête c, avec pour expression de test $c.\text{BEXP}$. Soit $\mathbf{i}$ le vecteur d'indices de c et a :

  - si a est dans la branche THEN,
    $\text{Exe}^s(a(\mathbf{i})) = \text{Exe}^s(c(\mathbf{i}))$ & $[\![c.\text{BEXP}]\!]@c(\mathbf{i})$
  - si a est dans la branche ELSE,
    $\text{Exe}^s(a(\mathbf{i})) = \text{Exe}^s(c(\mathbf{i}))$ & $\neg([\![c.\text{BEXP}]\!]@c(\mathbf{i}))$

2. De manière cohérente avec cet aspect séquentiel, dans ce qui suit, les conjonctions sont *associatives à gauche* : par exemple, $A \wedge B$ & $C \wedge D$ s'interprète comme : $((A \wedge B)$ & $C) \wedge D$. En outre, dans tous les cas où une expression C peut être *indéfinie* si une certaine autre expression A est *fausse*, nous viserons à ce que C apparaisse seulement dans des expressions $(... \wedge A \wedge ...)$ & $C...$, de sorte que les expressions qui en résultent soient toujours *définies*.

- a est immédiatement contenu dans une boucle DO ou PDO de tête c, les expressions de borne étant c.LB et c.UB respectivement. Soit $\mathbf{i}$ le vecteur d'indices de c et $\mathbf{i}::j$ le vecteur d'indices de a. Nous avons alors : $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i}::j)) = \mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i}))$ & $([\![c.\mathrm{LB}]\!]@c(\mathbf{i}) \leq j \leq [\![c.\mathrm{UB}]\!]@c(\mathbf{i}))$
- a est immédiatement contenu dans un WHILE de tête c. Soit $\mathbf{i}$ le vecteur d'indices de c et $\mathbf{i}::j$ le vecteur d'indices de a. Soit b la première instruction dans le corps du WHILE, c'est-à-dire l'instruction de test, d'expression b.BEXP.
  - Si a n'est pas b : $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i}::j)) = \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i}::j))$ & $([\![b.\mathrm{BEXP}]\!]@b(\mathbf{i}::j))$
  - Si a est b : $\mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i}::1)) = \mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i}))$ et, pour les autres itérations : $\mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i}::j)) = \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i}::(j-1)))$ & $([\![b.\mathrm{BEXP}]\!]@b(\mathbf{i}::(j-1)))$

Il est possible d'étendre l'expression de $\mathrm{Exe}^{\mathsf{s}}$ aux situations où le programme séquentiel boucle, en fonction, cependant, de la donnée explicite du WHILE sur lequel a lieu cette boucle infinie. Soit $c(\mathbf{i})$ la tête de l'instance de WHILE sur lequel a lieu cette boucle infinie, et $e(\mathbf{i})$ l'instance de ENDWHILE correspondante. Pour cette instance $e(\mathbf{i})$ et toutes les autres instances $a(\mathbf{j})$ qui lui succèdent dans l'ordre séquentiel, l'expression ci-dessus donnée de $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{j}))$, qui n'est d'ailleurs pas nécessairement définie dans ce cas, doit être remplacée par : $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{j})) = \mathit{faux}$.

La condition de terminaison du programme séquentiel, qui est bien entendu indécidable en général, peut alors être formellement exprimée ici. Utilisant les notations qui ont été introduites concernant les WHILE, nous obtenons :

Condition de terminaison séquentielle :
Pour toute boucle WHILE*, de tête c et de test b,*

$$\mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i})) \Rightarrow \exists\, j\ \ \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i}::j)) \ \&\ \neg([\![b.\mathrm{BEXP}]\!]@b(\mathbf{i}::j))$$

En cas de non terminaison séquentielle, c'est-à-dire de boucle infinie, l'implication ci-dessus est transgressée seulement sur l'instance de WHILE réalisant la boucle infinie.

3.2 Dépendances

Considérant deux instructions a et b, indicées par $\mathbf{i}$ et $\mathbf{j}$ respectivement, un prédicat $\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j}))$ exprimera que : "dans le cas où $a(\mathbf{i})$ et $b(\mathbf{j})$ sont exécutés tous deux dans la version séquentielle, dans cet ordre, alors ils accèdent tous deux à une même case mémoire, l'un au moins des deux écrivant sur celle-ci" [3].

3. Ecartons d'emblée le cas particulier où $a(\mathbf{i})$ et $b(\mathbf{j})$ sont une même instance, accédant à une même variable en lecture puis en écriture – par exemple une instance d'une ins-

La référence à la version séquentielle est cruciale ici, car nous allons toujours être intéressés par la préservation, dans la version parallèle, des dépendances *telles qu'elles apparaissent dans la version séquentielle*. En d'autres termes, la condition *"dépendance implique précédence"* doit s'interpréter comme : *"dépendance (apparaissant dans la version séquentielle) implique précédence (assurée dans la version parallèle)"*

Soient donc deux instances $a(\mathbf{i})$ et $b(\mathbf{j})$ comportant respectivement des références *expa* et *expb* à des variables (autres que des variables d'événements), ces deux références *n'étant pas toutes deux en lecture*. Dans le cas où $a(\mathbf{i})$ est exécuté dans la version séquentielle, nous notons $[expa@a(\mathbf{i})]^S$ la variable à laquelle *expa* fait référence lors de cette exécution. La relation $\equiv$ entre variables auxquelles il est fait deux références distinctes, signifie qu'elles sont la même variable. $\ll$ désignant l'ordre séquentiel, nous pouvons donner une expression de Dep :

$$\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j})) = (a(\mathbf{i}) \ll b(\mathbf{j})) \wedge ([expa@a(\mathbf{i})]^S \equiv [expb@b(\mathbf{j})]^S)$$

Cette expression de Dep n'est pas nécessairement définie lorsque $a(\mathbf{i})$ et(ou) $b(\mathbf{j})$ n'est pas exécuté dans la version séquentielle. Nous serons donc amené, dans ce qui suit, à faire figurer Dep dans des expressions :

$$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j})) \;\&\; \mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j}))$$

faisant usage de la conjonction séquentielle & introduite ci-dessus (§3.1).

Examinons un exemple d'un élément de programme supposé être exécuté (figure 3.1). La version séquentielle est donnée à droite de la figure. Nous regardons les dépendances relatives au tableau $A()$. Considérons d'abord le cas où l'on peut vérifier statiquement que la variable entière N n'est pas écrite entre `ps` et `c` dans la version séquentielle. (Une telle vérification peut être aisée dans un langage de type Fortran sans appels à des sous-programmes, mais plus ardue, voire intraitable, quand par exemple des pointeurs sont utilisés.) Alors, les instructions `a` et `b` font référence au même emplacement mémoire dans la version séquentielle (parce que $[\![N]\!]@a = [\![N]\!]@b$), et en écriture en ce qui concerne `a`. Par conséquent, nous avons une dépendance : $\mathrm{Dep}(a, b) = \mathit{vrai}$. Par ailleurs, nous n'avons pas de dépendance (au moins en ce qui concerne les références visibles ici) entre `a` et `c`.

Considérons à présent le cas plus difficile où nous ne pouvons pas savoir statiquement ce qui peut advenir de N entre `ps` et `c` dans la version séquentielle. Nous devrons alors supposer (dans une approximation conservatrice)

truction du genre "`x=x+1`". En vertu des spécifications de notre modèle d'exécution (§2.2) concernant l'exécution d'une instance, cette dépendance "réflexive" sera nécessairement préservée, dans notre sens, par la précédence "lecture précède écriture" garantie au cours de l'exécution d'une même instance.

```
          ...                              ...
ps:       psections                        continue
            section                          continue
              ...                              ...
a:            A(N)=...                         A(N)=...
              ...                              ...
            section                          continue
              ...                              ...
b:            ...=A(N)+...                     ...=A(N)+...
c:            ...=A(N+1)*...                   ...=A(N+1)*...
            section                          continue
              ...                              ...
n1:           N=...                            N=...
              ...                              ...
          endpsections                     continue
```

FIG. 3.1: *L'étude des dépendances : un exemple*

qu'il y a une dépendance de `a` vers `b` *et* de `a` vers `c` (sauf preuve contraire, nous pouvons très bien avoir $[\![N]\!]@a = [\![N]\!]@c + 1$!).

Sur cet exemple, considérons maintenant les relations de dépendance concernant la variable N. Cette variable est une sortie de `n1` et une entrée de `a`, `b` et `c`, d'où une relation de dépendance de `a` vers `n1`, de `b` vers `n1` et de `c` vers `n1`. Il n'y a pas de dépendances mutuelles entre `a`, `b` et `c`, concernant la variable N, car celle-ci y figure en lecture.

Comme le montre cet exemple, il sera souvent impossible (notamment en raison de la *référence dynamique de variables*) de spécifier statiquement les relations de dépendance exactes. Dans de tels cas, nous devrons rechercher une approximation conservatrice de ces dépendances, c'est-à-dire une approximation "par le dessus" (nous y reviendrons).

3.3 Précédences

Le prédicat de précédence $\mathrm{Pre}^{\mathsf{s}}$ exprime, rappelons-le, les relations de précédence qui s'appliqueraient dans le programme parallèle, en conséquence du modèle d'exécution, *en supposant* que les variables qui figurent dans la définition de ces relations reçoivent leurs valeurs "séquentielles". Nous allons caractériser $\mathrm{Pre}^{\mathsf{s}}$ à partir d'une *précédence de contrôle* Pre^{0} et d'une *précédence de synchronisation* $\mathrm{Sync}^{\mathsf{s}}$ (correspondant aux paires POST/WAIT). Il peut être intéressant de remarquer qu'*il peut exister plusieurs prédicats non équivalents exprimant correctement une relation de précédence.* Cela provient de la circonstance suivante : par un prédicat $\mathrm{Pre}(a, b)$, nous exprimons que *"dans le cas où* $a(\mathbf{i})$ *et* $b(\mathbf{j})$ *sont exécutés, celui-là s'exécute avant celui-ci*

(et de telle manière que les mises à jour de variables s'opèrent entretemps), mais nous ne sommes pas intéressés par ce qui est exprimé si l'une de ces instances n'est pas exécutée." Considérant deux instances d'instructions α et β, pour tout prédicat P exprimant correctement que α précède β dans le cas où tous deux sont exécutés, tout autre prédicat Q tel que :

$$\mathrm{Exe}(\alpha) \wedge \mathrm{Exe}(\beta) \wedge P \Rightarrow Q \Rightarrow (\mathrm{Exe}(\alpha) \wedge \mathrm{Exe}(\beta) \wedge P) \vee (\neg\,\mathrm{Exe}(\alpha) \vee \neg\,\mathrm{Exe}(\beta))$$

exprime tout aussi correctement cela. Cette même multiplicité de prédicats corrects se présente aussi pour les dépendances ; en outre, nous vérifierons aisément que la propriété de "préservation des dépendances" que nous allons considérer est (comme il se doit) invariante par tout changement de prédicats corrects.

Exprimons à présent les précédences de contrôle par un prédicat Pre^0 indépendant de l'exécution particulière du programme – en fait, un prédicat ne dépendant d'aucune variable. Nous nous intéresserons ensuite aux précédences de synchronisation et à la manière dont elles se combinent avec les précédences de contrôle.

3.3.1 Expression des précédences de contrôle

Calcul des précédences sur les vecteurs d'indices

Afin d'exprimer Pre^0, nous devons exprimer les précédences sur les vecteurs d'indices, notées $\prec$.

Soit $\mathbf{i}$ le vecteur d'indices d'une instruction ; soit k l'indice le plus intérieur de $\mathbf{i}$; soit $\mathbf{j}$ le vecteur d'indices "restant" (éventuellement vide), de telle sorte que $\mathbf{i}$ est la concaténation de $\mathbf{j}$ avec k, notée $\mathbf{i} = \mathbf{j}::k$.

- Si k est l'indice d'une boucle DO ou WHILE :

  $(\mathbf{i}_1 \prec \mathbf{i}_2) = (\mathbf{j}_1 \prec \mathbf{j}_2) \vee ((\mathbf{j}_1 = \mathbf{j}_2) \wedge (k_1 < k_2))$

- Si k est l'indice d'une boucle PDO :

  $(\mathbf{i}_1 \prec \mathbf{i}_2) = (\mathbf{j}_1 \prec \mathbf{j}_2)$

- *(Pour démarrer la récurrence :)* Si $\mathbf{j}$ est vide – notons $[\,]$ le vecteur d'indices vide –, nous posons :

  $([\,]_1 \prec [\,]_2) = \mathit{faux}$; $([\,]_1 = [\,]_2) = \mathit{vrai}$

Cela nous conduit directement à l'expression de $\mathrm{Pre}^0(a, a)$ pour une instruction a. Soit $\mathbf{i}$ le vecteur d'indices de a ; soit $a(\mathbf{i}_1)$ et $a(\mathbf{i}_2)$ deux instances de a ; nous avons :

$\mathrm{Pre}^0(a(\mathbf{i}_1), a(\mathbf{i}_2)) = (\mathbf{i}_1 \prec \mathbf{i}_2)$

Expression de Pre^0 *entre des instructions distinctes*

Soient a et b deux instructions telles que a figure avant b dans le texte du programme. Nous allons donner des expressions de $\mathrm{Pre}^0(a,b)$ dans les différents cas. Dans ce qui suit, nous n'avons pas besoin de distinguer le cas particulier où a et b sont dans deux branches alternatives d'un IF car, en raison de la précédente remarque (sur la multiplicité de prédicats corrects), la partie de $\mathrm{Pre}^0(a,b)$ correspondant à des instances mutuellement exclusives de a et b sera superflue.

Dans le cas où a et b ne sont pas dans la même boucle ni le même PSECTIONS, nous obtenons :
$\mathrm{Pre}^0(a,b) = \mathit{vrai}$; $\mathrm{Pre}^0(b,a) = \mathit{faux}$

Autrement, nous considérons la boucle ou le PSECTIONS le plus interne contenant à la fois a et b. Soit c la tête de cette instruction structurée, et $\mathbf{i}$ le vecteur d'indices de c.

- Si c est une tête de boucle, soit $\mathbf{j} = \mathbf{i}::h$ le vecteur d'indices commun à a et b. (h désigne l'indice de la boucle ; les vecteurs d'indices de a et b sont des concaténations de $\mathbf{j}$ avec des vecteurs disjoints (éventuellement vides) $\mathbf{k}$ et $\mathbf{l}$ respectivement.) Nous obtenons :
  $\mathrm{Pre}^0(a(\mathbf{j}_a::\mathbf{k}_a), b(\mathbf{j}_b::\mathbf{l}_b)) = (\mathbf{j}_a \prec \mathbf{j}_b) \vee (\mathbf{j}_a = \mathbf{j}_b)$
  $\mathrm{Pre}^0(b(\mathbf{j}_b::\mathbf{l}_b), a(\mathbf{j}_a::\mathbf{k}_a)) = (\mathbf{j}_b \prec \mathbf{j}_a)$

Dans les deux cas qui suivent, où c est une tête de PSECTIONS, le vecteur d'indices commun à a et b est $\mathbf{i}$. De nouveau, les vecteurs d'indices de a et b sont des concaténations de $\mathbf{i}$ avec des vecteurs d'indices disjoints (éventuellement vides) $\mathbf{k}$ et $\mathbf{l}$ respectivement.

- Si c est une tête de PSECTIONS et si a et b sont dans la même SECTION de cette PSECTIONS :
  $\mathrm{Pre}^0(a(\mathbf{i}_a::\mathbf{k}_a), b(\mathbf{i}_b::\mathbf{l}_b)) = (\mathbf{i}_a \prec \mathbf{i}_b) \vee (\mathbf{i}_a = \mathbf{i}_b)$
  $\mathrm{Pre}^0(b(\mathbf{i}_b::\mathbf{l}_b), a(\mathbf{i}_a::\mathbf{k}_a)) = (\mathbf{i}_b \prec \mathbf{i}_a)$
- Si c est une tête de PSECTIONS et si a et b sont dans des SECTIONS distinctes de cette PSECTIONS :
  $\mathrm{Pre}^0(a(\mathbf{i}_a::\mathbf{k}_a), b(\mathbf{i}_b::\mathbf{l}_b)) = (\mathbf{i}_a \prec \mathbf{i}_b)$
  $\mathrm{Pre}^0(b(\mathbf{i}_b::\mathbf{l}_b), a(\mathbf{i}_a::\mathbf{k}_a)) = (\mathbf{i}_b \prec \mathbf{i}_a)$

3.3.2 Combiner précédences de contrôle et de synchronisation

Pour obtenir la relation de précédence $\mathrm{Pre}^{\mathsf{s}}$, nous devons composer la précédence de contrôle Pre^0 et les relations de synchronisation $\mathrm{Sync}^{\mathsf{s}}$ réalisées

```
          psections
            section
p:            post E
            section
              if (B)
              then
w:              wait E
              else
                A=1
              endif
a:            A=2
          endpsections
```

FIG. 3.2: *Pourquoi la relation de précédence n'est pas fermée par transitivité*

à l'aide de paires POST/WAIT (nous allons examiner ces dernières dans un instant (§3.3.3) ; pour le moment, nous les supposons données). Cette composition de Pre^0 avec $\mathrm{Sync}^{\mathsf{s}}$ n'est pas exactement une fermeture transitive, comme on aurait pu le penser : $\mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), b(\mathbf{j})) \wedge \mathrm{Pre}^{\mathsf{s}}(b(\mathbf{j}), c(\mathbf{k}))$ n'implique pas nécessairement $\mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), c(\mathbf{k}))$. Il suffit, pour s'en convaincre, de considérer l'exemple donné par la figure 3.2. Dans cet exemple, nous supposons que l'événement E n'est posté nulle part ailleurs qu'indiqué. Nous le voyons, `p` précède `w` et `w` précède `a`, mais `p` ne précède pas `a`, car la branche ELSE peut être empruntée et, sans attendre E, `a` peut être exécuté concurremment avec `p`. Nous pouvons seulement dire que, si `w` est exécuté, alors `p` précède `a`.

Au lieu de la transitivité, nous avons donc une "transitivité modulo $\mathrm{Exe}^{\mathsf{s}}$" :

$$\mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), b(\mathbf{j})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j})) \wedge \mathrm{Pre}^{\mathsf{s}}(b(\mathbf{j}), c(\mathbf{k})) \Rightarrow \mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), c(\mathbf{k}))$$

Considérant le *graphe des précédences*, dont les nœuds sont les instances d'instructions et dont les arcs sont les liens de précédence Pre^0 et $\mathrm{Sync}^{\mathsf{s}}$, la relation $\mathrm{Pre}^{\mathsf{s}}$ sera obtenue par cette fermeture transitive modulo $\mathrm{Exe}^{\mathsf{s}}$ le long des chemins, et par disjonction entre des chemins alternatifs, selon un schéma de "conjonction en série, disjonction en parallèle", à partir de Pre^0 et de $\mathrm{Sync}^{\mathsf{s}}$. La fermeture transitive en ce qui concerne Pre^0 seul, est déjà assurée au travers des expressions de Pre^0 précédemment fournies. Par conséquent, les chemins de précédence à considérer pour obtenir $\mathrm{Pre}^{\mathsf{s}}$ font alterner les liaisons Pre^0 et $\mathrm{Sync}^{\mathsf{s}}$, de la manière suivante :

$$\alpha \to \pi_1 \rightsquigarrow \omega_1 \to \pi_2 \rightsquigarrow \omega_2 \to \ldots \to \pi_n \rightsquigarrow \omega_n \to \beta,$$

où $\to$ désigne la relation Pre^0, π_i désigne un POST, ω_i désigne un WAIT, et $\rightsquigarrow$ désigne la relation de synchronisation $\mathrm{Sync}^{\mathsf{s}}$.

Le calcul de $\mathrm{Pre}^{\mathsf{s}}$ sera réalisé à l'aide d'expressions telles que :

$$\mathrm{Pre}^0(\alpha,\pi_1) \wedge \mathrm{Exe}^{\mathsf{s}}(\pi_1) \wedge \mathrm{Sync}^{\mathsf{s}}(\pi_1,\omega_1) \wedge \mathrm{Exe}^{\mathsf{s}}(\omega_1) \wedge$$
$$\mathrm{Pre}^0(\omega_1,\pi_2) \wedge \ldots \wedge \mathrm{Sync}^{\mathsf{s}}(\pi_n,\omega_n) \wedge \mathrm{Exe}^{\mathsf{s}}(\omega_n) \wedge \mathrm{Pre}^0(\omega_n,\beta) \quad \Rightarrow \quad \mathrm{Pre}^{\mathsf{s}}(\alpha,\beta)$$

Plus exactement, ces expressions doivent être modifiées ainsi : comme $\mathrm{Sync}^{\mathsf{s}}(\pi_i,\omega_i)$ n'est pas nécessairement défini si π_i ou ω_i n'est pas exécuté (en raison des références dynamiques qui peuvent exister dans les tableaux d'événements), il faudrait utiliser ici la conjonction séquentielle précédemment introduite (§3.1) et écrire :

$$\mathrm{Exe}^{\mathsf{s}}(\pi_i) \wedge \mathrm{Exe}^{\mathsf{s}}(\omega_i) \;\&\; \mathrm{Sync}^{\mathsf{s}}(\pi_i,\omega_i)$$

au lieu de :

$$\mathrm{Exe}^{\mathsf{s}}(\pi_i) \wedge \mathrm{Sync}^{\mathsf{s}}(\pi_i,\omega_i) \wedge \mathrm{Exe}^{\mathsf{s}}(\omega_i)$$

Par commodité, nous pourrons, dans la suite, incorporer dans le prédicat $\mathrm{Sync}^{\mathsf{s}}(\pi,\omega)$ les conditions d'exécution de π et ω.

Reprenons le cheminement du graphe des précédences donné plus haut, alternant les $\rightarrow$ et les $\rightsquigarrow$, considéré pour la définition de $\mathrm{Pre}^{\mathsf{s}}(\alpha,\beta)$. Dans le cas où α est un POST, on doit aussi considérer des trajets décrits par ce cheminement où "$\alpha \rightarrow \pi_1$" est remplacé par "$\alpha = \pi_1$". Autrement dit, un chemin de précédence exprimé à travers $\mathrm{Pre}^{\mathsf{s}}$ peut commencer par une synchronisation. *En revanche*, dans le cas où β est un WAIT, *nous ne nous autorisons pas* à considérer des chemins dans lesquels "$\omega_n \rightarrow \beta$" est remplacé par "$\omega_n = \beta$". Autrement dit, un chemin de précédence exprimé à travers $\mathrm{Pre}^{\mathsf{s}}$ peut *commencer*, mais ne peut *se terminer*, par une synchronisation. Pourquoi introduisons-nous cette dissymétrie ? Nous avons été amené à le faire en raison du comportement très particulier des instructions WAIT.

Le problème des références dynamiques dans les WAIT

Dans notre langage, nous nous autorisons les références dynamiques dans les WAIT [4]. Cette faculté soulève des problèmes particuliers, illustrés par le petit exemple de la figure 3.3.

4. L'intérêt d'autoriser les références dynamiques dans les instructions de synchronisation est sans doute limité... Il ne semble cependant pas inexistant, comme le montrera un petit exemple plus loin (§6.4). Par ailleurs, cette faculté n'introduit pas de grandes difficultés dans nos démonstrations. *Cependant*, dans un premier temps, nous avions été amené à imposer une condition de référence statique dans les WAIT. Il est apparu (récemment) que la restriction sur la définition des précédences que nous venons de mentionner – le fait qu'un chemin du graphe de précédence intervenant dans la définition de Pre ou $\mathrm{Pre}^{\mathsf{s}}$ ne puisse pas se terminer par une synchronisation – simplifiait quelque peu nos démonstrations, en même temps qu'elle permettait, sans grande complication supplémentaire, de lever cette condition de référence statique.

```
n1:   N=1
      psections
        section
          ...
n2:       N=2
p:        post E(N)
          ...
        section
          ...
w:        wait E(N)
          ...
      endpsections
```

FIG. 3.3: *Un cas de référence dynamique dans un* WAIT

Dans cet exemple, nous supposons que N ne figure que comme indiqué. L'intention sous-jacente est d'obtenir une synchronisation entre `p` et `w`, associée à l'événement $E(2)$. De fait, *selon la sémantique séquentielle*, N a bien la valeur 2 en `p` et en `w`. *Cependant*, dans une exécution parallèle, N peut être égal à 1 lorsque le contrôle arrive à `w`, qui attendra alors l'événement $E(1)$ (alors que `p` postera toujours $E(2)$). La dépendance $\mathrm{Dep}(n2, w)$ associée à la variable N n'est donc pas préservée, comme on aurait pu le souhaiter, par le chemin du graphe de précédence $n2 \to p \rightsquigarrow w$ (en utilisant les notations ci-dessus). C'est pourquoi nous avons défini notre prédicat $\mathrm{Pre}^{\mathsf{s}}$ de manière à exclure de tels chemins de précédence se terminant par une synchronisation. Cette restriction que nous apportons nous permet, à la lumière du modèle d'exécution (§2.2), d'assurer pour le prédicat $\mathrm{Pre}^{\mathsf{s}}$ la propriété suivante : si α est une instance exécutée et w une instance de WAIT en attente, $\mathrm{Pre}^{\mathsf{s}}(\alpha, w)$ implique que α s'exécute avant que w entre en attente (et de telle manière que les mises à jour de variables aient lieu entretemps). Ainsi, dans notre exemple, si $\mathrm{Pre}^{\mathsf{s}}(n2, w)$ était tout de même assuré, grâce à un chemin de précédence n'apparaissant pas ici, alors l'écriture de N en `n2` précéderait (au sens de nos précédences) sa lecture lors de l'entrée en attente de `w`, qui se mettrait alors en attente de $E(2)$ (à moins que N ait été réécrit entretemps).

3.3.3 Les précédences de synchronisation

La relation de synchronisation entre un POST et un WAIT correspondant est beaucoup moins simple à considérer que la précédence de contrôle Pre^{0}.

La première difficulté provient du fait que nous autorisons la référence dynamique de variable dans les instructions POST, WAIT et CLEAR. Cette latitude, jointe au fait que le prédicat d'exécution Exe n'est défini qu'en référence à une exécution donnée du programme parallèle, implique que les

synchronisations POST/WAIT qui se réalisent effectivement au cours d'une exécution donnée du programme parallèle, avec la précédence qui y est associée selon le modèle d'exécution des POST et WAIT (§2.2 et 2.3), dépendent *a priori* de l'exécution considérée. Se donnant une telle exécution parallèle, soit ω une instance de WAIT qui y est exécutée. Il existe alors au moins une instance de POST π (et peut-être plusieurs) dont l'exécution, postant l'événement correspondant, a rendu possible l'exécution de ω et assuré la précédence de synchronisation correspondante entre π et ω. Nous notons $\mathrm{Sync}(\pi, \omega)$ le prédicat, dépendant donc de l'exécution parallèle considérée, qui exprime les précédences de synchronisation ainsi réalisées.

Le prédicat Pre de précédence propre à une exécution parallèle donnée, s'obtient alors à partir de Pre^0 et Sync, exactement selon les mêmes formules que $\mathrm{Pre}^{\mathsf{s}}$ à partir de Pre^0 et $\mathrm{Sync}^{\mathsf{s}}$ (§3.3.2).

Nous nous heurtons donc, pour Sync et par conséquent pour Pre, à ce problème de "référence à une exécution particulière" que nous avons déjà rencontré pour le prédicat Exe, et qui ne se présente pas pour Pre^0. Par conséquent, dans notre préoccupation de nous référer à la seule sémantique séquentielle, nous devrons considérer, plutôt que cette relation Sync (que nous serions d'ailleurs bien en peine d'*observer* sur une exécution parallèle !), une relation $\mathrm{Sync}^{\mathsf{s}}$ décrivant les synchronisations qui se présentent en supposant que les variables intervenant dans les références d'événements reçoivent leurs valeurs "séquentielles". Rappelons ici que, lorsque nous considérons la version séquentielle de notre programme, les synchronisations y sont désactivées, mais nous gardons trace des événements auxquels elles font référence (§2.3). C'est ce qui nous autorisera, ci-après, à considérer "l'exécution de synchronisations dans la version séquentielle".

La seconde difficulté est d'une nature différente – et elle va nous conduire à préciser certaines propriétés que nous prescrirons concernant les synchronisations. Par un prédicat $\mathrm{Sync}^{\mathsf{s}}(\pi, \omega)$ entre une instance de POST π et une instance de WAIT ω correspondante, nous voulons exprimer que *"supposant la sémantique séquentielle, si π et ω sont exécutés tous deux, alors nécessairement π est exécuté avant ω."*. Cela présuppose qu'aucun autre POST ne soit susceptible de déclencher l'exécution de ω, en postant le même événement. De fait, dans une situation où, toujours sous la sémantique séquentielle, plusieurs instances de POST non mutuellement exclusives apparaîtraient susceptibles de déclencher l'exécution d'une même instance d'un WAIT, aucune précédence ne serait alors garantie entre l'un quelconque de ces POST et ce WAIT – dans une telle situation, nous n'aurions pas de relation $\mathrm{Sync}^{\mathsf{s}}$ entre l'un quelconque de ces POST et ce WAIT – et une telle situation serait intraitable dans le cadre où nous nous plaçons. (Remarquons que, réciproquement, un POST peut très bien poster vers plusieurs WAIT : cela ne nous pose pas de problème ici.)

Dans la mesure où une seule instance de POST doit pouvoir déclencher

une instance de WAIT dans une exécution donnée, il apparaît raisonnable que deux instances de POST faisant référence au même événement ne s'exécutent pas en parallèle. Il apparaît naturel, également, de prescrire qu'une instance de CLEAR faisant référence au même événement qu'un POST ou qu'un WAIT ne se trouve pas en conflit mémoire avec ce dernier.

Nous allons exprimer ces restrictions à travers deux hypothèses, qui ne sont pas indépendantes l'une de l'autre, concernant l'utilisation des instructions de synchronisation. Ces hypothèses sont assez générales ; elles suffiront pour le résultat que nous démontrerons plus loin. *Cependant*, faisant usage des précédences $\mathrm{Pre}^{\mathsf{s}}$, elles présupposent que les relations de synchronisation $\mathrm{Sync}^{\mathsf{s}}$ sont données, ce qui constitue une faiblesse pour deux raisons : nous ne fournissons pas une expression générale de ces relations $\mathrm{Sync}^{\mathsf{s}}$; en outre, ces deux hypothèses concourent précisément à l'existence même de ces relations de synchronisation, ce qui induit une circularité.

Par la suite, cependant, nous allons donner des expressions de $\mathrm{Sync}^{\mathsf{s}}$ et de ces hypothèses dans un cas restreint (mais encore très général).

Hypothèse S1 (Absence de conflit mémoire sur les synchronisations) *Soient deux instances d'instructions de synchronisation (*POST*,* WAIT *ou* CLEAR*)* θ *et* ω*, toutes deux exécutées dans la version séquentielle d'une instance de programme, et y faisant référence à une même variable événement* ε*. Hormis le cas où l'une est un* POST *et l'autre est un* WAIT*, et le cas où toutes deux sont des* WAIT*,* θ *et* ω *sont reliés par une précédence* $\mathrm{Pre}^{\mathsf{s}}$*.*

Commentaire Cette hypothèse paraît manifestement raisonnable. Fondamentalement, elle se ramène bien à une "absence de conflit mémoire" portant sur des variables événements, si nous considérons le fait que les WAIT lisent, et les POST et CLEAR écrivent sur ces variables [5].

Hypothèse S2 (Précédence assurée entre POST **et** WAIT**)** *Supposant l'hypothèse S1, soit une instance de* WAIT γ *exécutée dans la version séquentielle d'une instance de programme. Il y a au plus une instance de* POST π*, exécutée dans la version séquentielle de cette instance de programme, remplissant les conditions suivantes :*

– π *et* γ *font référence à la même variable événement* ε *dans cette version*

5. On peut trouver insolite l'exclusion ici faite de conflits mémoire CLEAR/CLEAR, étant donné que deux CLEAR en conflit écrivent la même valeur *cleared*, sans conséquence apparente sur les synchronisations (au contraire de conflits POST/POST)... En fait, les démonstrations qui suivent pourraient être adaptées au cas où l'on voudrait admettre de tels conflits CLEAR/CLEAR (auquel cas il faudrait faire la supposition que deux écritures concurrentes de la *même valeur* sur la même case mémoire se traduisent par la présence, dans cette case, de *cette* valeur).

séquentielle.

- *il n'y a pas d'instance de* CLEAR θ*, exécutée et faisant référence à* ε *dans cette version séquentielle, telle que les précédences* $\mathrm{Pre}^{\mathsf{s}}$ *impliquées par l'hypothèse S1 soient de* π *vers* θ *et de* θ *vers* γ*.*

- *nous n'avons pas* $\mathrm{Pre}^{\mathsf{s}}(\gamma, \pi)$*.*

Commentaire Cela exprime qu'au plus une instance de POST est *susceptible de déclencher* l'exécution d'une instance de WAIT, dans l'instance de programme donnée. Cependant, cette instance de POST peut dépendre de l'instance de programme considérée, c'est-à-dire des valeurs des paramètres ; par exemple, il arrivera souvent que deux POST postant le même événement se trouvent dans deux branches alternatives d'un IF : ce n'est pas contraire à notre hypothèse car ces deux POST sont mutuellement exclusifs.

Nous allons maintenant, comme annoncé, nous intéresser à une version restreinte de ces hypothèses S1 et S2. Nous allons considérer le cas où les relations de précédence mentionnées dans les énoncés ci-dessus de S1 et S2 sont des précédences de contrôle Pre^0, et non des précédences plus générales $\mathrm{Pre}^{\mathsf{s}}$. De la sorte, l'énoncé de ces hypothèses ainsi restreintes ne dépend plus de la donnée préalable des synchronisations $\mathrm{Sync}^{\mathsf{s}}$.

Sous cette restriction, nous allons pouvoir donner une expression de $\mathrm{Sync}^{\mathsf{s}}$. Selon des notations déjà introduites (§3.2), pour toute instance d'instruction POST, WAIT ou CLEAR γ exécutée dans la version séquentielle, nous notons $[\varepsilon_\gamma]^S$ la variable événement à laquelle γ fait référence dans la version séquentielle.

Pour une instance de POST π et une instance de WAIT γ, définissons un prédicat $\mathrm{Sync}^\star$ ainsi :

$$\begin{aligned}\mathrm{Sync}^\star(\pi,\gamma) \quad = \quad & \mathrm{Exe}^{\mathsf{s}}(\pi) \wedge \mathrm{Exe}^{\mathsf{s}}(\gamma)\ \&\ ([\varepsilon_\pi]^S \equiv [\varepsilon_\gamma]^S) \wedge \neg\,\mathrm{Pre}^0(\gamma,\pi) \wedge \\ & (\forall \text{ instance de CLEAR } \theta, \\ & \mathrm{Exe}^{\mathsf{s}}(\gamma) \wedge \mathrm{Exe}^{\mathsf{s}}(\theta)\ \&\ ([\varepsilon_\theta]^S \equiv [\varepsilon_\gamma]^S) \\ & \Rightarrow \mathrm{Pre}^0(\theta,\pi) \vee \mathrm{Pre}^0(\gamma,\theta)\ \ )\end{aligned}$$

$\mathrm{Sync}^\star(\pi,\gamma)$ exprime que π est susceptible de déclencher l'exécution de γ, dans notre sens.
Alors, sous les restrictions que nous venons de mentionner, concernant les hypothèses S1 et S2, cette dernière peut être exprimée comme suit : pour toutes instances de POST π_i et toute instance de WAIT γ :

$$\mathrm{Sync}^\star(\pi_1,\gamma) \wedge \mathrm{Sync}^\star(\pi_2,\gamma) \Rightarrow \pi_1 = \pi_2$$

auquel cas $\mathrm{Sync}^{\star}$ exprime en fait la relation de synchronisation $\mathrm{Sync}^{\mathsf{s}}$ qui nous intéresse.

Il est important de conserver à l'esprit que nos hypothèses, aussi bien sous leur forme "générale" que sous leur forme "restreinte", font référence à la sémantique de la seule version séquentielle, et ne dépendent aucunement d'une exécution particulière du programme parallèle.

Chapitre 4

Quelques résultats préliminaires

Avant de démontrer notre théorème 1 dans le prochain chapitre, nous allons établir quelques résultats préliminaires.

4.1 Approximations conservatrices des prédicats

A la lumière du théorème que nous allons démontrer, la vérification de la propriété d'équivalence sémantique sur un programme parallèle donné, exigera de vérifier l'implication suivante :

$$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j})) \;\&\; \mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j})) \Rightarrow \mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), b(\mathbf{j}))$$

pour toutes les instances d'instructions $a(\mathbf{i})$ et $b(\mathbf{j})$ du programme considéré. (La présence ici de la conjonction séquentielle & a été expliquée dans le §3.2.)

Dans de nombreux cas, il sera impossible de produire des expressions exactes des prédicats figurant ici. Il faudra alors rechercher des *approximations conservatrices* de ces prédicats, c'est-à-dire des approximations telles que leur utilisation en lieu et place des prédicats exacts ne conduise jamais à donner une conclusion positive quand la propriété d'équivalence n'est pas vérifiée – mais peut conduire à une absence de conclusion dans des situations où la propriété est, en fait, vérifiée.

La direction de l'implication ci-dessus rend claires les approximations qui seront conservatrices : ce seront des approximations "par le dessus" pour $\mathrm{Exe}^{\mathsf{s}}$ et Dep, par le dessous pour $\mathrm{Pre}^{\mathsf{s}}$: nous considérerons donc des prédicats $\mathrm{Exe}^{\mathsf{s}\star}$, $\mathrm{Dep}^{\star}$ et $\mathrm{Pre}^{\mathsf{s}}_{\star}$ tels que $\mathrm{Exe}^{\mathsf{s}} \Rightarrow \mathrm{Exe}^{\mathsf{s}\star}$, $\mathrm{Dep} \Rightarrow \mathrm{Dep}^{\star}$ et $\mathrm{Pre}^{\mathsf{s}}_{\star} \Rightarrow \mathrm{Pre}^{\mathsf{s}}$, prédicats signifiant respectivement que "une instance d'instruction risque d'être exécutée", "une dépendance risque d'exister" et "une précédence doit exister".

Le calcul d'une relation $\mathrm{Pre}^{\mathsf{s}}$ utilise des prédicats Pre^{0}, $\mathrm{Sync}^{\mathsf{s}}$ et $\mathrm{Exe}^{\mathsf{s}}$, dans une "fermeture transitive modulo $\mathrm{Exe}^{\mathsf{s}}$ le long de chemins", comme nous

l'avons vu précédemment. Pre^0 sera assez facilement calculable ; le calcul de $\mathrm{Sync}^{\mathsf{s}}$ peut être plus compliqué. Par conséquent, l'approximation de $\mathrm{Pre}^{\mathsf{s}}$ par le dessous peut nécessiter d'approximer $\mathrm{Exe}^{\mathsf{s}}$ par le dessous, par un prédicat $\mathrm{Exe}^{\mathsf{s}}_\star$ tel que $\mathrm{Exe}^{\mathsf{s}}_\star \Rightarrow \mathrm{Exe}^{\mathsf{s}}$; et de considérer seulement une partie des chemins du graphe des précédences.

4.2 Un lemme sur le prédicat d'exécution

Il va nous être utile de préciser dans quels cas, et dans quel sens, la condition $\mathrm{Exe}(\alpha)$ d'exécution d'une instance d'instruction α dans une exécution parallèle donnée, dépend strictement de l'exécution d'autres instances d'instructions β telles que $\mathrm{Pre}^0(\beta, \alpha)$. C'est l'objet du lemme 1 ci-après.

On note $\psi(\alpha)$ la condition (dépendant de l'exécution parallèle considérée) pour que α soit exécutée *ou* bloquée en attente *ou* bloquée en suspens (rappelons que α est dite *atteinte* dans les deux premiers cas). Si α est l'instance (unique) de la première instruction du programme, $\psi(\alpha) = \mathit{vrai}$. Pour les autres instances d'instructions du programme, nous avons le :

Lemme 1 *Considérant un programme parallèle, pour toute exécution de ce programme, et pour toute instance α de toute instruction autre que la première instruction du programme, $\psi(\alpha)$ est pleinement déterminé par l'exécution d'une ou plusieurs instance(s) d'instruction(s) β telle(s) que $\mathrm{Pre}^0(\beta, \alpha)$. Une partie au moins de ces instances β sont spécifiées indépendamment de l'exécution considérée du programme ; les autres, s'il y en a, sont spécifiées par l'exécution des premières. Si l'une au moins de ces instances β est en erreur d'exécution, cela entraîne $\psi(\alpha) = \mathit{faux}$.*

Nous avons $\mathrm{Exe}(\alpha) = \psi(\alpha)$ sauf éventuellement dans les trois cas suivants :

- *α est une instance d'un* WAIT *w : alors, $\psi(\alpha)$ exprime la condition pour que α soit atteinte (ou la condition pour que α soit atteinte ou bloquée en suspens, dans le cas où w est à la fois un* WAIT *et la première instruction dans le corps d'un* PDO *ou dans une* SECTION *d'un* PSECTIONS*). Sous cette condition, cependant, α peut être bloquée en attente (ou bloquée en suspens), au lieu de s'exécuter, dans une situation de blocage (ou de boucle infinie dans le deuxième cas).*

- *α est une instance de la première instruction dans le corps d'un* PDO*, ou dans une* SECTION *d'un* PSECTIONS*, sans être une instance d'un* WAIT *: alors, $\psi(\alpha)$ exprime la condition pour α d'être exécutée ou bloquée en suspens ; cette dernière possibilité peut survenir en cas de blocage ou de boucle infinie.*

- *α est une instance d'un* ENDWHILE *: alors, $\psi(\alpha)$ exprime la condition pour α d'être exécutée ou bloquée en suspens ; cette dernière possibilité*

survient lorsque le WHILE *boucle indéfiniment, ou en cas de blocage ou de boucle infinie dans une itération de ce* WHILE.

La démonstration de ce lemme, laborieuse mais sans difficultés, est donnée en annexe A.

Le principal intérêt de ce lemme réside dans l'observation précédemment faite que Pre^0 est indépendant de l'exécution particulière du programme parallèle que l'on considère. Sa signification peut être résumée ainsi : à l'exception des instructions WAIT, et de quelques autres susceptibles d'être en suspens, l'exécution d'une instance d'instruction dans une certaine exécution du programme, dépend d'instances qui *sont vouées à s'exécuter* avant elle (par la structure de contrôle du programme), et non pas d'instances qui, simplement, *se trouveraient s'exécuter* avant elle dans une certaine exécution du programme, comme impliqué par la simple causalité.

4.3 Une notion de date d'exécution

Pour démontrer le théorème 1, nous devons établir l'existence, sur une exécution quelconque d'un programme parallèle, d'une *fonction de date d'exécution*[1] possédant certaines propriétés. C'est l'objet du résultat suivant :

Lemme de temps discrétisé : *Pour toute exécution d'un programme parallèle, nous considérons les précédences* Pre *et* Sync *associées à cette exécution (§3.3.3). Supposons les quatre propriétés suivantes :*

i. A chaque instance d'instruction α exécutée, est associé un intervalle de temps physique $[t_\alpha, t'_\alpha]$, avec $t'_\alpha \geq t_\alpha$, appelé intervalle d'exécution *de α.*

ii. Lorsque les intervalles d'exécution de deux instances d'instructions α et β s'intersectent et que α calcule une variable x correspondant à une entrée de β, la valeur écrite par α n'est pas *disponible comme entrée pour β.*

iii. En conséquence de (ii), pour toutes instances α et β exécutées, si $\mathrm{Pre}(\alpha, \beta)$*, ou bien si* $\mathrm{Sync}(\alpha, \beta)$*, alors $t_\beta > t'_\alpha$.*

iv. Pour tout instant t, seules un nombre fini d'instances α ont commencé de s'exécuter avant t.

A toute instance d'instruction α exécutée au cours de cette exécution du programme, peut être associé un entier positif $\tau(\alpha)$, appelé la date d'exécution *de α, ayant les propriétés suivantes :*

1. $\tau(\alpha)$ est fonction croissante de t_α.

1. ...ou *fonction de pas de temps d'exécution.* L'appellation "date d'exécution" a été suggérée par François Thomasset.

2. Causalité calculatoire : *pour toutes instances d'instructions α et β exécutées, une valeur calculée par α ne peut pas être utilisée comme entrée par β sauf si $\tau(\beta) > \tau(\alpha)$.*

3. *En conséquence, pour toutes instances d'instructions α et β exécutées, si* $\mathrm{Pre}(\alpha, \beta)$*, ou bien si* $\mathrm{Sync}(\alpha, \beta)$*, nous avons $\tau(\beta) > \tau(\alpha)$.*

4. *τ est* causalement défini*, c'est-à-dire que $\tau(\alpha)$ dépend seulement des intervalles d'exécution de α et des instances commençant leur exécution avant α.*

Commentaire : Les hypothèses *(i)* et *(ii)* peuvent être interprétées ainsi : pour toute instance α venant à être exécutée, α reçoit ses entrées (le cas échéant) à l'instant t_α ou peu avant, puis s'exécute sans aucune entrée/sortie jusqu'à un instant t'_α auquel (ou peu de temps après lequel) α fournit ses résultats (le cas échéant). (Nous pouvons éventuellement avoir $t_\alpha = t'_\alpha$, notamment lorsque α ne fait pas de calcul.) Ces hypothèses sont justifiées ici par les spécifications de notre modèle d'exécution (§2.2) concernant l'exécution d'une instance. L'hypothèse *(iii)* est une conséquence de la condition de mise à jour de variables incluse dans la relation Pre (§2.3). L'hypothèse *(iv)* a une signification seulement quand le programme boucle indéfiniment, une situation que nous voulons considérer aussi ; cette hypothèse est alors justifiée par la finitude des ressources disponibles (particulièrement le nombre fini de processus).

Mentionnons aussi que, en cohérence avec notre définition de l'"exécution" dans le cas d'une instance d'un WAIT, l'intervalle d'exécution d'une telle instance ne contient pas le temps d'attente ; il ne commence que lorsque la valeur *posted* de l'événement auquel elle fait référence a été détectée.

Preuve : Considérant une exécution du programme parallèle, ordonnons l'ensemble (dénombrable) des instances d'instructions α exécutées, dans l'ordre croissant des instants initiaux t_α. Lorsque plusieurs instants initiaux sont égaux, nous rangeons les instances correspondantes arbitrairement. En raison de *(iv)*, cet ordre des instances est *bien fondé* : ces instances sont donc ordonnées selon une suite. Cette suite est infinie lorsque l'exécution du programme boucle indéfiniment.

Cette suite d'instances, finie ou infinie, sera notée $\alpha_1, \alpha_2, \ldots \alpha_N, \ldots$. Pour la commodité, l'intervalle d'exécution de α_i sera noté $[t_i, t'_i]$.

La date τ est définie par la procédure suivante :

1. Poser $\tau(\alpha_1) = 1$ et $i = 1$

2. Pour les entiers j suivant i, s'il y en a, tels que α_j existe et $t_j \leq \min(t'_k \mid i \leq k < j)$, poser $\tau(\alpha_j) = \tau(\alpha_i)$

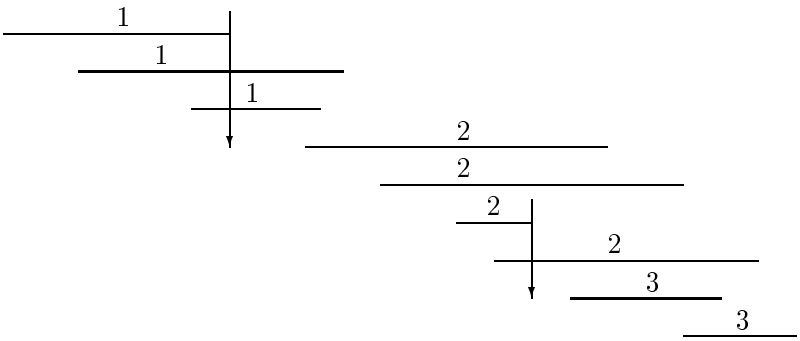

FIG. 4.1: *Définition de la date d'exécution : un petit exemple*

3. Si la suite des instances α n'est pas encore épuisée, soit j l'indice du premier α restant. Poser $\tau(\alpha_j) = \tau(\alpha_i) + 1$; poser $i = j$ et aller à [2.]

Il est simple de vérifier que la fonction τ ainsi définie satisfait aux propriétés demandées : nous remarquons que les instances d'instructions associées à la même date ont des intervalles d'exécution qui s'intersectent : par conséquent, *(ii)* implique la propriété de causalité calculatoire ; en outre, deux instances dont les intervalles d'exécution sont disjoints – notamment deux instances qui sont dans une relation de précédence Pre – ont des dates différentes. ◀

La figure 4.1 montre un petit exemple d'application de la procédure qui définit la date d'exécution. Le temps s'y écoule de gauche à droite ; les intervalles d'exécution des 9 premières instances y sont représentés dans l'ordre prescrit (celui des instants initiaux, de haut en bas sur la figure). La date assignée (de 1 à 3 ici) est inscrite sur chaque segment correspondant. Le point 2 de la procédure attribue une même date aussi longtemps que le segment considéré intersecte tous les précédents segments de même date. Lorsque ce n'est plus le cas, la procédure passe au point 3 et incrémente la date (flèches verticales sur la figure).

Insistons bien sur le fait que nous allons faire usage du seul fait qu'une fonction de date *existe* ; nous n'allons pas avoir besoin de la *calculer effectivement*. D'ailleurs, la fonction de date dépend grandement de l'exécution particulière du programme que l'on est en train de considérer, même lorsque le programme parallèle est sémantiquement équivalent à sa version séquentielle.

En ce qui concerne la sémantique des programmes, des instances associées à la même date seront considérées comme si elles s'exécutaient "en même temps". C'est ainsi que, pour la commodité, nous dirons que "telle instance

s'exécute à telle date". La causalité calculatoire (propriété 2) est fondamentale ici : elle assure que le résultat d'un calcul "effectué à telle date" n'est pas disponible avant la date suivante. (La propriété 4 de définition causale ne sera pas utilisée ici : c'est un sous-produit de notre résultat.) Pour ainsi dire, ce que nous considérons ici, c'est une *discrétisation en temps préservant la causalité*.

La réciproque de la propriété 2 n'est pas vraie : avoir $\tau(\beta) > \tau(\alpha)$ n'implique pas qu'une valeur calculée par α peut être utilisée comme entrée par β (d'ailleurs, il se peut que les intervalles d'exécution de α et β s'intersectent, comme illustré par le petit exemple de la figure). Mais il faut remarquer que cette disponibilité est assurée si l'on a $\mathrm{Pre}(\alpha, \beta)$ (du moment, bien sûr, que la variable n'a pas été réécrite entretemps) ; ou $\mathrm{Sync}(\alpha, \beta)$ (auquel cas, d'ailleurs, α écrit sur un événement lu par β).

Le besoin d'introduire une notion de date d'exécution est la raison principale pour laquelle nous considérons des instructions *simples* plutôt que des instructions *structurées* : il est fréquent qu'une instruction structurée s'étende sur plusieurs dates, dans notre sens. Cela n'est pas dû à la durée d'exécution de cette instruction structurée, mais plutôt à l'existence d'échanges entrée/sortie *durant* l'intervalle d'exécution. Ainsi, un appel à un sous-programme se verra attribuer *une* date, si longue que puisse être la durée de son exécution, dès lors que les entrées/sorties surviennent seulement au commencement et à la fin de l'appel[2].

D'autres notions de date

La notion de date d'exécution que nous introduisons ici présente certaines ressemblances, et certaines différences, avec la célèbre notion de *temps linéaire* proposée par Lamport [26, 35]. Lamport considère des *processus*, sur chacun desquels se succèdent des *événements*. Parmi ces événements, peuvent figurer l'envoi d'un message vers un autre processus, ou la réception d'un message venant d'un autre processus. Les processus sont supposés s'influencer seulement à travers ces échanges de messages.

A chaque processus est associée une horloge permettant de dater, par un nombre entier, chaque événement se produisant successivement sur ce processus : le passage d'un événement au suivant incrémente l'horloge locale du processus ; par ailleurs, chaque message échangé étant accompagné de la date de son émission (pour l'horloge locale du processus émetteur), sa réception entraîne l'incrémentation de l'horloge locale du processus récepteur, si nécessaire, de manière que la date (locale au processus récepteur) de la réception soit supérieure à la date (locale au processus émetteur) de l'émission.

2. Dans la même veine, dans certains langages, comme *C* par exemple, où plusieurs opérations peuvent être condensées en une instruction apparemment "simple", une telle instruction peut avoir à être considérée comme structurée dans notre sens, et son exécution s'étendra alors sur plusieurs dates.

Le schéma de Lamport permet (moyennant la contrainte – inévitable dans ce contexte – que tout message échangé doit contenir la donnée d'une date) de réaliser effectivement une "horloge globale" à partir seulement d'horloges locales à chaque processus, et exprimant la causalité, au sens suivant : un événement x ne peut influencer un événement y (par le biais de messages éventuellement) que si le temps associé à x est inférieur strictement au temps associé à y. Cet aspect de causalité est évidemment ce qui rapproche le temps linéaire ainsi conçu de "notre" date. Une différence importante entre notre schéma et celui de Lamport – ou certains autres schémas plus complexes décrits dans [35] – c'est que ces derniers visent à permettre de calculer *effectivement* des temps permettant de dater des événements, tandis que, nous l'avons dit, il nous suffit de connaître *l'existence* d'une fonction de date $\tau()$ vérifiant nos propriétés ; nous n'aurons nul besoin d'un protocole permettant de calculer effectivement $\tau()$.

4.4 L'exécution monoprocessus ordonnée

Une exécution **monoprocessus** est une exécution du programme parallèle, obtenue lorsqu'il y a un seul processus disponible. Comme nous l'avons mentionné (§2.2), il sera exigé qu'un programme parallèle ne se bloque pas, quel que soit le nombre de processus disponibles. Par conséquent, une exécution monoprocessus ne doit pas se bloquer.

Nous allons nous intéresser à l'**exécution monoprocessus ordonnée**, définie comme l'exécution monoprocessus pour laquelle la condition d'assignation séquentielle (§2.2) s'étend à toutes les constructions parallèles, et non seulement à celles contenant des synchronisations. Sous l'hypothèse supplémentaire que cette exécution monoprocessus ordonnée ne se bloque pas – une hypothèse dans notre théorème – son comportement correspond exactement à celui de la version séquentielle, non seulement du point de vue de l'équivalence sémantique, mais aussi en ce qui concerne l'ordre d'exécution des instances d'instructions.

Par ailleurs, si l'exécution monoprocessus ordonnée se bloque, son comportement correspond à celui de la version séquentielle, dans le même sens, *jusqu'à* l'instance de WAIT sur laquelle se produit le blocage.

Il peut être intéressant d'exprimer formellement la condition de non blocage de l'exécution monoprocessus ordonnée. En vertu de la remarque ci-dessus, nous devons exprimer que *toute instance de* WAIT *exécutée dans la version séquentielle n'est pas un* WAIT *bloquant, supposant que l'exécution monoprocessus ordonnée se poursuit jusqu'à ce point, avec équivalence sémantique jusque-là.* Utilisant cette équivalence sémantique, nous pouvons exprimer la condition de non blocage ($\ll$ exprime toujours l'ordre d'exécution séquentielle) :

Condition de non blocage pour une exécution monoprocessus ordonnée :[3]

$$
\begin{array}{l}
\forall \text{ instance de } \textsc{wait}\ \gamma, \\
\qquad \begin{array}{rcl}
\mathrm{Exe}^{\mathsf{s}}(\gamma) & \Rightarrow & \exists \text{ instance de } \textsc{post}\ \pi, \mathrm{Exe}^{\mathsf{s}}(\gamma) \wedge \mathrm{Exe}^{\mathsf{s}}(\pi) \\
 & & \wedge(\pi \ll \gamma)\ \&\ ([\varepsilon_\pi]^S \equiv [\varepsilon_\gamma]^S) \\
 & & \wedge(\ \forall \text{ instance de } \textsc{clear}\ \theta, \\
 & & \mathrm{Exe}^{\mathsf{s}}(\gamma) \wedge \mathrm{Exe}^{\mathsf{s}}(\theta)\ \&\ ([\varepsilon_\theta]^S \equiv [\varepsilon_\gamma]^S) \\
 & & \Rightarrow (\theta \ll \pi \vee \gamma \ll \theta\ )\ \ )
\end{array}
\end{array}
$$

3. En vertu de ce que nous avons vu (§3.1) sur l'expression formelle de $\mathrm{Exe}^{\mathsf{s}}$, cette condition est formellement plus satisfaisante sous l'hypothèse que l'exécution monoprocessus ordonnée ne boucle pas indéfiniment (également une hypothèse de notre théorème), car dans le cas contraire, l'expression de $\mathrm{Exe}^{\mathsf{s}}$ dépend de la donnée explicite de la localisation où se situe la boucle infinie.

Chapitre 5

Le théorème d'équivalence sémantique

Nous allons maintenant démontrer notre théorème d'équivalence sémantique. Soit donc un programme parallèle écrit dans le langage que nous avons défini (chap.2). Nous considérons la version séquentielle de ce programme, que nous supposons se conformer aux règles de notre langage ayant trait aux programmes séquentiels (entre autres, la condition de détermination, l'absence d'erreur d'exécution). Moyennant certaines propriétés portant sur la version séquentielle ou sur la version "monoprocessus ordonnée" qui, nous venons de le voir (§4.4), lui est très proche, nous allons démontrer l'équivalence sémantique entre notre programme parallèle et sa version séquentielle.

Théorème 1 *Sous les hypothèses suivantes :*

i. Hypothèses S1 et S2 (données au §3.3.3) ;

ii. Pas de blocage en attente dans l'exécution monoprocessus ordonnée ;

iii. Pas de boucle infinie dans l'exécution monoprocessus ordonnée ;

iv. Pour toutes instances d'instructions $a(\mathbf{i})$ *et* $b(\mathbf{j})$,
$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j}))$ & $\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j})) \Rightarrow \mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), b(\mathbf{j}))$;

le programme parallèle est sémantiquement équivalent à sa version séquentielle. En particulier, aucune exécution parallèle ne peut se bloquer (en erreur ou en attente) ni boucler indéfiniment.

Preuve : Nous considérons une instance de programme, obtenue en donnant des valeurs aux paramètres. Alors, il y a une seule exécution monoprocessus ordonnée de ce programme, que nous noterons **S**, tandis qu'il y a en général beaucoup d'exécutions possibles de la version parallèle. Nous allons

considérer l'une d'entre elles, que nous noterons **P**. Dans la suite de cette démonstration, les prédicats Exe et Pre, définis au chapitre 3, sont donc les prédicats d'exécution et de précédence associés à cette exécution parallèle **P**.

Nous allons d'abord démontrer l'équivalence sémantique étendue à toutes les instances d'instructions exécutées dans cette exécution parallèle **P**, et à toutes les variables auxquelles ces instances font référence **(points 1 à 4)** ; puis nous prouverons que, inversement, toutes les instances exécutées dans l'exécution monoprocessus ordonnée **S** le sont également dans cette exécution parallèle **P (point 5)**.

Point 1 : Nous nous intéressons à la *fonction de date d'exécution* associée, selon la construction que nous avons exposée (§4.3), à l'exécution parallèle **P** que nous considérons.

Soit τ une date qui vérifie l'hypothèse de récurrence suivante :

> *Equivalence sémantique jusqu'à la date* $\tau - 1$ incluse : Pour toute instance d'instruction α exécutée avant τ dans **P**, α est aussi exécutée dans **S**. Toute référence figurant dans α en entrée (y compris les références d'événements) désigne la même variable dans les deux exécutions, et cette variable a été écrite par la même autre instance dans les deux exécutions. Corrélativement, toutes les variables écrites, dans **P**, avant la date τ (y compris les variables d'événements), ont fait l'objet des mêmes calculs, effectués par les mêmes instances, dans le même ordre, dans **S**.

Cette hypothèse exprime bien une équivalence sémantique entre **P** et **S** sur toutes les entrées et les sorties de toutes les instances d'instructions exécutées avant τ dans **P**. (Par voie de conséquence, notamment, **P** ne produit pas d'erreur d'exécution avant τ.)

Nous souhaitons montrer que cette équivalence sémantique s'étend à la date τ. Comme elle s'applique clairement au démarrage du programme, cela assurera l'équivalence sémantique le long de toute l'exécution parallèle **P**.

En premier lieu, nous devons introduire un lemme.

Lemme 2 *Nous supposons l'hypothèse d'équivalence sémantique jusqu'à la date* $\tau - 1$*, et une instance d'instruction* γ *exécutée à* τ *dans* **P**. *Pour toute instance d'instruction* α *telle que l'on ait* $\mathrm{Exe}^{\mathbf{S}}(\alpha)$ *et* $\mathrm{Pre}^{\mathbf{S}}(\alpha, \gamma)$*, nous avons* $\mathrm{Exe}(\alpha)$ *et* $\mathrm{Pre}(\alpha, \gamma)$.

Ce résultat s'applique également si, au lieu de γ*, on considère une instance d'instruction exécutée avant* τ *dans* **P**.

La démonstration de ce lemme est donnée en annexe B.

Nous devons montrer que, pour toute instance d'instruction γ venant à être exécutée à la date τ dans **P**, l'équivalence sémantique se propage à γ. Tout d'abord, nous allons montrer que l'équivalence sémantique s'étend aux *entrées* de γ. Pour ce faire, nous serons amené à distinguer le cas où une telle entrée n'est pas un événement **(point 2)**, puis le cas où elle en est un **(point 3)**. Ensuite, nous observerons que cette exécution de γ ne produit pas d'erreur d'exécution, puis nous montrerons que l'équivalence sémantique s'étend aux *sorties* de γ, en établissant qu'il n'y a pas de conflit mémoire entre instances s'exécutant à la date τ **(point 4)**. Ainsi sera établie l'équivalence sémantique jusqu'à la date τ.

Nous considérons donc une instance d'instruction γ venant à être exécutée à la date τ dans **P**. Selon le lemme 1 (§4.2 ou annexe A), le fait que γ soit *atteint ou bloqué en suspens* (ceci n'impliquant pas qu'il soit exécuté) est entièrement déterminé par une ou plusieurs instance(s) d'instruction(s) β vérifiant $\mathrm{Pre}^0(\beta, \gamma)$, et qui ont été exécutées (par conséquent avant τ, car Pre^0 est commun à toutes les exécutions). En raison de l'hypothèse de récurrence (équivalence sémantique jusqu'à la date $\tau - 1$), ces mêmes instances β sont exécutées dans **S**, et déterminent identiquement que γ est atteint ou bloqué en suspens dans **S**. Par conséquent, γ s'exécute dans **S** (selon les hypothèses *(ii)* et *(iii)*, aucune instance ne demeure *en attente* ni *en suspens* dans **S**) : nous avons $\mathrm{Exe}^{\mathsf{s}}(\gamma)$.

Considérons une référence ξ figurant dans γ en entrée. En vue d'établir l'équivalence sémantique pour cette entrée, étant donné que nous supposons l'équivalence sémantique jusqu'à la date $\tau - 1$, nous devons exclure deux possibilités :

1. la possibilité que la référence ξ dans γ ne désigne pas la même variable dans **S** ; ou, dans le cas contraire (soit alors x la variable que désigne ξ dans les deux exécutions),

2. la possibilité que la valeur de x utilisée par γ comme entrée ne soit pas obtenue par les mêmes calculs dans les deux exécutions.

Nous montrons d'abord, par une récurrence sur l'ordre d'indirection (§2.1) de ξ, qu'éliminer la possibilité 1 revient à éliminer la possibilité 2. La possibilité 1 ne peut se présenter si l'ordre d'indirection de ξ est 0, car une telle référence désigne statiquement une même variable dans toute exécution. Maintenant, si ξ a pour ordre d'indirection $n > 0$, supposons que les possibilités 1 et 2 aient été éliminées pour toutes les entrées de γ d'ordre d'indirection inférieur à n. Alors, l'équivalence sémantique s'applique à toutes ces entrées, donc notamment à toutes les références contenues dans la liste d'expressions d'indices de ξ. Par conséquent, ξ désigne la même variable x dans les deux exécutions, et il suffit alors d'éliminer la possibilité 2 pour cette entrée x.

Considérant donc une référence ξ désignant la même variable x dans les deux exécutions, nous devons éliminer la possibilité 2 en nous assurant que la valeur de x utilisée par γ comme entrée a été calculée de la même manière dans les deux exécutions.

Point 2 : Considérons d'abord le cas où x n'est pas un événement.

Soit β l'instance d'instruction qui calcule la valeur de x utilisée par γ comme entrée dans **S**. β existe, en raison de la condition de détermination (§2.3). Nous allons d'abord montrer que nous avons $\mathrm{Exe}(\beta)$ et $\mathrm{Pre}(\beta,\gamma)$, ce qui impliquera que β est exécuté dans **P** avant τ. L'hypothèse de récurrence impliquera alors que x est calculé de la même manière par β dans les deux exécutions, et la précédence ainsi obtenue impliquera que cette valeur de x est disponible comme entrée pour γ dans **P** – *à moins que* quelque autre calcul de x puisse interférer entre β et γ, une éventualité que nous exclurons ensuite.

Nous avons $\mathrm{Exe}^{\mathbf{s}}(\beta)$ parce que c'est β qui calcule x pour γ dans **S** ; nous avons vu que $\mathrm{Exe}^{\mathbf{s}}(\gamma)$; nous avons $\mathrm{Dep}(\beta,\gamma)$ car β calcule une variable (qui n'est pas un événement) utilisée par γ *dans* **S**, l'exécution en référence à laquelle Dep est défini. Par conséquent, selon l'hypothèse *(iv)*, nous avons $\mathrm{Pre}^{\mathbf{s}}(\beta,\gamma)$. Selon le lemme 2, $\mathrm{Exe}^{\mathbf{s}}(\beta) \wedge \mathrm{Pre}^{\mathbf{s}}(\beta,\gamma)$ implique que β s'exécute dans **P** et que l'on a $\mathrm{Pre}(\beta,\gamma)$. Maintenant, afin de confirmer que l'équivalence sémantique s'étend à l'entrée x de γ, nous devons montrer que, dans **P**, β calcule bien x pour γ sans qu'une autre instance d'instruction δ, calculant également x dans **P**, interfère entre β et γ.

Une telle instance δ s'exécuterait avant τ, car sa sortie x serait disponible pour γ ; donc, en raison de l'hypothèse de récurrence, δ s'exécuterait aussi dans **S** et y calculerait la même variable x. Il y aurait donc une dépendance Dep entre β et δ, qui, avec $\mathrm{Exe}^{\mathbf{s}}(\beta)$ et $\mathrm{Exe}^{\mathbf{s}}(\delta)$, impliquerait, selon *(iv)*, une précédence $\mathrm{Pre}^{\mathbf{s}}$ entre β et δ. Dans quel sens cette précédence serait-elle ? Si nous avions $\mathrm{Pre}^{\mathbf{s}}(\delta,\beta)$, le lemme 2 appliqué à β impliquerait alors $\mathrm{Pre}(\delta,\beta)$, ce qui, avec $\mathrm{Pre}(\beta,\gamma)$, empêcherait que la valeur de x calculée par δ soit lue par γ. Donc, nous aurions $\mathrm{Pre}^{\mathbf{s}}(\beta,\delta)$: δ s'exécuterait bien après β dans **S**. Comme β calcule bien x pour γ dans **S**, une telle circonstance impliquerait que δ s'exécute *après* γ dans **S**. Si tel était le cas, nous aurions $\mathrm{Dep}(\gamma,\delta)$, ce qui, avec $\mathrm{Exe}^{\mathbf{s}}(\delta)$ et $\mathrm{Exe}^{\mathbf{s}}(\gamma)$, impliquerait $\mathrm{Pre}^{\mathbf{s}}(\gamma,\delta)$. δ s'exécuterait dans **P** avant τ, donc le lemme 2 s'appliquerait à δ : $\mathrm{Exe}^{\mathbf{s}}(\gamma) \wedge \mathrm{Pre}^{\mathbf{s}}(\gamma,\delta)$ impliquerait $\mathrm{Pre}(\gamma,\delta)$, ce qui contredirait le fait que δ s'exécute avant τ dans **P**. Une telle instance δ n'existe donc pas.

Point 3 : Considérons maintenant le cas où x, variable d'entrée de γ dans **S** et **P**, est un événement. Cela implique que γ est un WAIT.

Selon l'hypothèse de récurrence, tous les calculs de x avant τ sont identiques dans les deux exécutions. Soit ρ l'instance d'instruction qui a modifié en dernier l'événement x dans cette histoire commune. Comme γ s'exécute à la date τ dans **P**, ρ existe et est un POST (et non un CLEAR).

Soit π l'instance d'instruction qui écrit sur x pour γ dans **S**. Comme γ est exécuté dans **S**, π est un POST et (π, γ) est une paire de synchronisation ; nous avons $\mathrm{Sync}^{\mathbf{S}}(\pi, \gamma)$.

Nous devons montrer que π est ρ. Cela terminera notre point 3. Si ce n'était le cas, les hypothèses S1 et S2 (§3.3.3) impliqueraient, ou bien $\mathrm{Pre}^{\mathbf{S}}(\gamma, \rho)$ (auquel cas ρ s'exécuterait *après* γ dans **S**), ou bien l'existence d'une instance de CLEAR θ faisant référence à l'événement x dans **S**, telle que $\mathrm{Pre}^{\mathbf{S}}(\rho, \theta) \wedge \mathrm{Pre}^{\mathbf{S}}(\theta, \gamma)$. Ecartons successivement ces deux éventualités.

> *L'éventualité* $\mathrm{Pre}^{\mathbf{S}}(\gamma, \rho)$. Comme ρ s'exécute avant τ dans **P**, le lemme 2 s'applique : $\mathrm{Exe}^{\mathbf{S}}(\gamma) \wedge \mathrm{Pre}^{\mathbf{S}}(\gamma, \rho)$ impliquerait $\mathrm{Pre}(\gamma, \rho)$, ce qui contredirait le fait que ρ s'exécute avant γ dans **P**.
>
> *L'éventualité* $\mathrm{Pre}^{\mathbf{S}}(\rho, \theta) \wedge \mathrm{Pre}^{\mathbf{S}}(\theta, \gamma)$. Selon le lemme 2, $\mathrm{Exe}^{\mathbf{S}}(\theta)$ et $\mathrm{Pre}^{\mathbf{S}}(\theta, \gamma)$ impliquerait $\mathrm{Pre}(\theta, \gamma)$: θ s'exécuterait avant τ dans **P**. Mais alors, $\mathrm{Pre}^{\mathbf{S}}(\rho, \theta)$ impliquerait une précédence similaire $\mathrm{Pre}(\rho, \theta)$ dans **P** (lemme 2 appliqué à θ) : x recevrait la valeur *cleared* par θ entre ρ et γ dans **P**, ce qui contredirait le fait que ρ est la dernière instance d'instruction écrivant sur x avant τ.

Point 4 : Nous avons établi que, considérant notre hypothèse de récurrence (l'équivalence sémantique jusqu'à la date $\tau - 1$) et une instance d'instruction γ exécutée à τ dans l'exécution parallèle **P** que nous considérons, γ s'exécute aussi dans **S** et l'équivalence sémantique s'étend à toutes les références d'entrée de γ : toute telle référence ξ désigne la même variable (notée x) dans les deux exécutions, et x contient la même valeur, calculée de manière similaire, à l'exécution de γ, dans **P** et **S**.

Cette équivalence portant sur les références d'entrée implique que toute référence de sortie η de γ désigne la même variable (notée y) dans les deux exécutions. Ainsi, dans les deux exécutions, γ effectue les mêmes calculs, sur les mêmes entrées, produisant les mêmes sorties. Par conséquent, l'exécution de γ dans **P** n'y produit pas d'erreur d'exécution. Pour s'assurer que l'équivalence sémantique s'étend aux sorties de γ à la date τ, il suffit de vérifier qu'il n'y a pas de conflit mémoire entre des instances d'instructions γ_i, par exemple γ_1 et γ_2, venant à être exécutées à la date τ dans **P**. En raison de l'équivalence sémantique sur les références d'entrée, les variables d'entrée et les références de sortie de γ_1 et γ_2, un tel conflit se refléterait aussi dans **S**.

Dans le cas où les références en conflit ne seraient pas des références d'événements, ce conflit se refléterait à travers une dépendance Dep, par exemple $\mathrm{Dep}(\gamma_1, \gamma_2)$, ce qui, selon l'hypothèse *(iv)*, impliquerait $\mathrm{Pre}^{\mathbf{S}}(\gamma_1, \gamma_2)$. Le lemme 2 s'applique : $\mathrm{Exe}^{\mathbf{S}}(\gamma_1)$ et $\mathrm{Pre}^{\mathbf{S}}(\gamma_1, \gamma_2)$ impliquerait $\mathrm{Pre}(\gamma_1, \gamma_2)$, ce qui contredit l'hypothèse que γ_1 et γ_2 s'exécutent tous deux à la date τ.

Dans le cas où les références en conflit seraient des références à un événement, tout d'abord, γ_1 et γ_2 ne seraient pas tous deux des WAIT : l'une au

moins des deux serait une écriture (pour qu'il y ait conflit), c'est-à-dire un CLEAR ou un POST. Deux cas sont à considérer.

- Le cas où γ_1 et γ_2 sont un POST et un WAIT, par exemple dans cet ordre, faisant référence (dans **S** et **P**) à un événement ε. Notons alors π l'instance de POST qui écrit sur ε pour γ_2 dans **S**. Nous avons $\mathrm{Sync}^{\mathbf{s}}(\pi, \gamma_2)$. Nous avons vu (point 3) que π s'exécute avant τ dans **P**, donc est différent de γ_1. Nous allons montrer que cela conduit à une contradiction. Selon les hypothèses S1 et S2 (§3.3.3), nous aurions, ou bien $\mathrm{Pre}^{\mathbf{s}}(\gamma_2, \gamma_1)$, ou bien l'existence d'une instance de CLEAR θ faisant référence à l'événement ε dans **S**, telle que $\mathrm{Pre}^{\mathbf{s}}(\gamma_1, \theta) \wedge \mathrm{Pre}^{\mathbf{s}}(\theta, \gamma_2)$. Ecartons successivement ces deux éventualités.

  L'éventualité $\mathrm{Pre}^{\mathbf{s}}(\gamma_2, \gamma_1)$. Comme γ_1 s'exécuterait à la date τ dans **P**, le lemme 2 s'applique : $\mathrm{Exe}^{\mathbf{s}}(\gamma_2) \wedge \mathrm{Pre}^{\mathbf{s}}(\gamma_2, \gamma_1)$ impliquerait $\mathrm{Pre}(\gamma_2, \gamma_1)$, ce qui contredirait le fait que γ_2 s'exécuterait à la date τ, comme γ_1, dans **P**.

  L'éventualité $\mathrm{Pre}^{\mathbf{s}}(\gamma_1, \theta) \wedge \mathrm{Pre}^{\mathbf{s}}(\theta, \gamma_2)$. Selon le lemme 2, ayant $\mathrm{Exe}^{\mathbf{s}}(\theta)$ et $\mathrm{Pre}^{\mathbf{s}}(\theta, \gamma_2)$ impliquerait que θ s'exécuterait avant τ dans **P**. Mais alors, $\mathrm{Pre}^{\mathbf{s}}(\gamma_1, \theta)$ impliquerait une précédence similaire dans **P** (lemme 2 appliqué à θ) : γ_1 s'exécuterait avant τ dans **P**, ce qui contredit l'hypothèse que γ_1 s'exécuterait à la date τ.

- Dans les autres cas, appliquons l'hypothèse S1. Nous avons une précédence $\mathrm{Pre}^{\mathbf{s}}$ entre γ_1 et γ_2, par exemple $\mathrm{Pre}^{\mathbf{s}}(\gamma_1, \gamma_2)$. Selon le lemme 2, ayant $\mathrm{Exe}^{\mathbf{s}}(\gamma_1)$ et $\mathrm{Pre}^{\mathbf{s}}(\gamma_1, \gamma_2)$ impliquerait $\mathrm{Pre}(\gamma_1, \gamma_2)$, ce qui contredirait le fait que γ_1 et γ_2 s'exécutent tous deux à la date τ.

Point 5 : Nous avons donc montré que toute instance d'instruction exécutée dans une quelconque exécution parallèle **P** est aussi exécutée dans **S**, et que toute variable figurant dans cette instance fait l'objet des mêmes calculs (et reçoit donc les mêmes valeurs) dans les deux exécutions jusqu'au dernier point atteint dans **P**.

Cette exécution parallèle **P** ne peut donc pas produire une boucle infinie : en effet, si une construction WHILE bouclait indéfiniment dans **P**, l'équivalence sémantique le long de **P** impliquerait une boucle infinie similaire dans **S**, ce qui est exclu par *(iii)*. Cette même équivalence sémantique le long de **P** assure que **P** ne produit pas d'erreur d'exécution.

Il reste à prouver que, réciproquement, toute instance d'instruction exécutée dans **S** s'exécute aussi dans toute exécution parallèle **P**. Supposons, par contradiction, qu'il y ait des instances exécutées dans **S** et non dans une certaine exécution parallèle **P** que nous considérons. Notons alors γ la première dans l'ordre séquentiel.

Appliquons le lemme 1 à l'exécution de γ dans **S**. Selon les hypothèses *(ii)* et *(iii)*, il n'y a pas d'instance d'instruction *bloquée en attente* ni *bloquée en suspens* dans **S**. Donc selon le lemme 1, $\mathrm{Exe}^{\mathsf{s}}(\gamma)$ dépend d'instances β qui sont en précédence Pre^0 avec γ et sont toutes exécutées dans **S**, avant γ. Par définition de γ, ces β s'exécutent dans **P**, avec équivalence sémantique, comme nous l'avons montré. Par conséquent, le lemme 1 et la non exécution de γ dans **P** impliquent que, dans **P**, ou bien γ est bloqué en suspens, ou bien γ est un WAIT bloqué en attente. Examinons ces deux éventualités.

En l'absence de boucle infinie et d'erreur d'exécution dans **P**, un blocage en suspens ne peut être provoqué que par un blocage en attente ailleurs. Plus précisément, la condition d'assignation séquentielle (§2.2) implique qu'une instance d'instruction ne peut être bloquée en suspens que si une certaine instance de WAIT, ω, située avant elle dans l'ordre séquentiel, est atteinte et bloquée en attente dans **P**. Etant atteint, ω est également atteint, et s'exécute, dans **S**. Son blocage dans **P**, et le fait qu'il précède γ dans l'ordre séquentiel, contredirait la définition de γ.

Il reste enfin le cas où γ est un WAIT bloqué en attente. Soit ε_γ l'événement auquel γ fait référence dans **S**. Il nous faut d'abord montrer que, dans **P**, γ fait référence à ce même événement.

Soit éventuellement x une variable autre que ε_γ figurant comme entrée de γ dans **S** (x existe en cas de référence dynamique). Par définition de γ, toutes les instances précédant γ dans **S** sont exécutées, avec équivalence sémantique, dans **P**. C'est donc le cas pour l'instance β_x qui calcule x pour γ dans **S** (pour chaque x éventuel, β_x existe, en raison de la condition de détermination (§2.3)).

Se peut-il que, dans **P**, x soit réécrit après β_x ? Si c'était le cas, ce serait par une instance notée δ_x. Alors, δ_x s'exécute dans **S**, avec équivalence sémantique, donc après γ. Nous aurions donc $\mathrm{Dep}(\gamma, \delta_x)$ (dépendance associée à x), $\mathrm{Exe}^{\mathsf{s}}(\gamma)$, $\mathrm{Exe}^{\mathsf{s}}(\delta_x)$, par conséquent $\mathrm{Pre}^{\mathsf{s}}(\gamma, \delta_x)$ selon *(iv)*. Selon le lemme 2 appliqué à δ_x, ayant $\mathrm{Exe}^{\mathsf{s}}(\gamma)$ et $\mathrm{Pre}^{\mathsf{s}}(\gamma, \delta_x)$ impliquerait $\mathrm{Exe}(\gamma)$, ce qui contredit le blocage sur γ dans **P**. Donc, x n'est pas écrit dans **P** après β_x.

Nous allons utiliser ici un lemme :

Lemme 3 *Nous supposons la propriété d'équivalence sémantique le long de* **P**. *Soit une instance de* WAIT γ *bloquée en attente dans* **P**. *Pour toute instance d'instruction* α *telle que l'on ait* $\mathrm{Exe}^{\mathsf{s}}(\alpha)$ *et* $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$, *nous avons* $\mathrm{Exe}(\alpha)$ *et* $\mathrm{Pre}(\alpha, \gamma)$.

La démonstration de ce lemme est donnée en annexe B, après celle du lemme 2.

Pour chaque éventuelle variable x, nous avons $\mathrm{Exe}^{\mathsf{s}}(\beta_x)$ et $\mathrm{Dep}(\beta_x, \gamma)$. Nous avons $\mathrm{Exe}^{\mathsf{s}}(\gamma)$. Donc, selon *(iv)*, nous avons $\mathrm{Pre}^{\mathsf{s}}(\beta_x, \gamma)$. Selon le lemme 3, nous avons donc $\mathrm{Pre}(\beta_x, \gamma)$.

Jointe au fait que x n'est pas réécrit dans **P** après β_x, cette précédence $\mathrm{Pre}(\beta_x, \gamma)$, reliant une instance exécutée à un WAIT en attente, implique (§3.3.2) que, *si* x est une variable d'entrée de γ dans **P** (ce que nous n'avons pas encore vérifié), la valeur de x lue par γ est (et demeure) celle calculée par β_x dans **P** comme dans **S**. La vérification que x est bien une entrée de γ dans **P** s'effectue par un argument de récurrence sur l'ordre d'indirection de x, analogue à celui que nous avons déjà produit dans le point 1. Nous trouvons donc ici que l'événement auquel γ fait référence dans **P** est et demeure ε_γ, comme dans **S**.

Par définition de γ, toutes les instances précédant γ dans **S** sont exécutées, avec équivalence sémantique, dans **P**. C'est donc le cas pour l'instance de POST, π, qui poste ε_γ pour γ dans **S**. Le blocage sur γ impliquerait donc que, dans **P**, une certaine instance de CLEAR, θ, s'exécute et donne la valeur *cleared* à ε_γ après π, et avant que γ l'utilise. θ s'exécute aussi dans **S** et donne la valeur *cleared* au même événement (comme nous l'avons vu, en vertu de l'équivalence sémantique étendue à toutes les instances exécutées dans **P**).

L'hypothèse S1 (§3.3.3) implique qu'il y a une relation $\mathrm{Pre}^{\mathsf{s}}$ entre θ et π, et entre θ et γ. Nous ne pouvons avoir $\mathrm{Pre}^{\mathsf{s}}(\theta, \pi)$ car cela impliquerait (lemme 2 appliqué à π) que θ s'exécute avant π dans **P**. Donc, nous aurions $\mathrm{Pre}^{\mathsf{s}}(\pi, \theta)$. Nous ne pouvons avoir $\mathrm{Pre}^{\mathsf{s}}(\theta, \gamma)$ car, combiné à $\mathrm{Pre}^{\mathsf{s}}(\pi, \theta)$, cela impliquerait un ordre d'exécution dans **S** : π avant θ avant γ, et π ne posterait pas pour γ, comme nous l'avons supposé. Donc nous aurions $\mathrm{Pre}^{\mathsf{s}}(\gamma, \theta)$.

θ s'exécute dans **P**, donc le lemme 2 s'applique à θ : γ s'exécute dans **S** et nous avons $\mathrm{Pre}^{\mathsf{s}}(\gamma, \theta)$; donc, γ s'exécute dans **P**, avant θ, ce qui contredit le blocage sur γ.

Cela termine la preuve de notre théorème.

◀

Les cas de blocage ou de boucle infinie

Dans la ligne de notre démonstration, il est assez simple de voir ce qui peut se produire lorsque l'exécution monoprocessus ordonnée **S** se bloque en attente, contrairement à l'hypothèse *(ii)*, ou boucle indéfiniment, contrairement à l'hypothèse *(iii)* – en gardant à l'esprit que ces deux éventualités sont *indésirables* dans notre contexte.

Lorsque **S** boucle indéfiniment, les instances d'instructions qui y sont exécutées, le sont également dans toute exécution parallèle **P** – d'où une boucle infinie dans cette exécution – mais, dans le cas où cette boucle infinie est englobée dans une construction parallèle, certaines de ses unités de travail peuvent s'exécuter dans **P**, alors qu'elles ne sont jamais atteintes dans **S**.

Lorsque **S** se bloque en attente, les instances d'instructions qui y sont exécutées, le sont également dans toute exécution parallèle **P**, mais, dans le cas (habituel) où l'instance de WAIT bloquante se situe dans une construction parallèle, il peut advenir que certaines de ses unités de travail s'exécutent dans **P**, bien qu'elles ne soient jamais atteintes dans **S**. Il peut même arriver que l'événement correspondant au WAIT bloquant se trouve posté ainsi, par une instance de POST située après l'instance de WAIT dans l'ordre séquentiel. Une telle circonstance se produira alors de manière *aléatoire*, dépendant essentiellement du nombre de processus disponibles et de l'assignation des unités de travail à ces processus, deux aspects de l'exécution du programme sur lesquels l'utilisateur est supposé n'avoir aucune prise.

Chapitre 6

Applications du théorème

Comme nous l'avons mentionné, une application de ce théorème a été développée au CERMICS [36, 4, 5]. Ce n'était pas l'objet du présent travail d'en développer d'autres, mais quelques indications utiles peuvent être présentées ici.

6.1 L'application développée au CERMICS

La thèse de Thierry Salset [36] présente une application de notre théorème, sous la forme d'un algorithme de vérification sur un sous-ensemble du langage que nous avons considéré ici. Outre [36], on se reportera utilement, pour une présentation succincte, à [5], reproduit ci-après, en annexe C.

Le sous-ensemble considéré est défini à travers certaines restrictions syntaxiques : pas de boucles dynamiques (WHILE) ; références statiques des variables (les expressions d'indices de tableaux étant de plus *linéaires*) ; les expressions de bornes de boucles DO et PDO contiennent seulement des paramètres du programme, des indices de boucles englobantes et des constantes numériques.

L'algorithme proposé vise à vérifier la préservation des dépendances, c'est-à-dire l'hypothèse *(iv)* (la principale) du théorème.

Le texte du programme à tester est partitionné en *blocs* d'instructions. Ce partitionnement obéit aux règles suivantes : une instruction WAIT est la première instruction du bloc où elle figure ; une instruction POST est la dernière instruction du bloc où elle figure ; un bloc ne peut contenir à la fois un WAIT et un POST. *De la sorte*, les précédences intérieures à un bloc sont seulement des précédences de contrôle qui se traitent aisément, et la description des relations de précédence entre instructions *se condense* commodément en des relations de précédence entre blocs, représentées par des arcs de contrôle associés à Pre^0 et des arcs de synchronisation associés à $\mathrm{Sync}^{\mathsf{s}}$ (*graphe de blocs* – ce partitionnement en blocs généralise les blocs de base décrits dans [1], et s'inspire de [3]).

Les arcs de contrôle et de synchronisation du graphe de blocs sont *étiquetés par des formules logiques* associant les instances de blocs reliées par les précédences en question. A ce point, il nous faut nous reporter à un exemple volontairement simple : celui des figures 1 et 2 de l'annexe C.

La figure 2 représente le découpage en blocs de l'élément de programme représenté par la figure 1, avec notamment deux blocs : celui contenant le WAIT, noté k_w, et celui de dessous consistant en le POST, noté k_p ; et deux arcs du graphe de blocs associé : l'arc de synchronisation s, de k_p vers k_w, étiqueté par la relation exprimant quelle instance x de k_p est ainsi synchronisée avec quelle instance y de k_w : $\mathrm{Sync}_{p,w}(x, y) \;=\; (x = y - 1)$; et, en sens contraire, l'arc de contrôle c, de k_w vers k_p, pareillement étiqueté par la relation associant les instances ainsi en précédence : $\mathrm{Pre}^0_{w,p}(y, x) \;=\; (y = x)$. Considérant alors la dépendance à préserver, entre les instructions a et b (figure 1) : $\mathrm{Dep}(a_x, b_y) \;=\; (y = x + 1)$, l'algorithme essaye les chemins de précédence reliant les blocs k_w et k_p, et trouve, sur l'exemple, que la dépendance est préservée (à travers le chemin cs). Les opérations effectuées sont montrées dans l'annexe C. Une autre illustration de telles opérations est exposée ci-après, dans l'exemple traité dans le §6.4.

Cet algorithme présenté dans [36, 5] permet, dans de nombreux cas, de donner une condition de préservation des dépendances sous la forme de relations entre paramètres du programme. Certaines améliorations récentes ([36], chap.8) permettent de traiter efficacement certains cas où une dépendance est préservée à travers, non pas *un* chemin de précédences comme dans l'exemple simple ci-dessus, mais un ensemble (non borné) de chemins, caractérisé par une expression régulière.

6.2 Quelques considérations de complexité

Revenons aux applications de notre théorème dans le cas général. Il s'avère très difficile d'évaluer la complexité (du point de vue algorithmique) que nous devons attendre d'un algorithme qui mettrait en application notre théorème en vue de vérifier des programmes parallèles. Nous pouvons seulement donner ici quelques indications.

Dans la vérification des préservations de dépendances, ce sont les synchronisations, non les précédences de contrôle, qui posent problème (Cette constatation, d'ailleurs, contribue à l'intérêt de la structure de blocs que nous venons de mentionner ci-avant (§6.1)). En effet, lorsque deux instances d'instructions α et β sont dans une précédence de contrôle $\mathrm{Pre}^0(\alpha, \beta)$ (ce qui est aisément vérifiable car, nous l'avons vu, les précédences Pre^0 sont calculables très facilement), une éventuelle relation de dépendance entre ces deux instances est nécessairement dans le "bon" sens, de α vers β dans l'exemple. En effet, une éventuelle dépendance $\mathrm{Dep}(\beta, \alpha)$ supposerait nécessairement que β précède α dans l'ordre séquentiel, ce qui est incompatible avec $\mathrm{Pre}^0(\alpha, \beta)$.

Nous ne devons donc nous préoccuper de la préservation d'une dépendance que dans la situation où il n'y a pas précédence de contrôle entre les instances considérées, ce qui suppose que les instructions correspondantes figurent dans une même construction parallèle. Il ne sera donc nécessaire de rechercher des dépendances qu'entre des instructions figurant dans une même construction parallèle, entre des instances de ces instructions figurant dans des unités de travail distinctes de cette construction parallèle.

Dans cette dernière situation, les éventuelles précédences qui préserveront les dépendances en question, feront nécessairement intervenir des synchronisations. La complexité des vérifications en question dépend donc très directement de la complexité du traitement des synchronisations.

En vertu du théorème 1, la vérification de correction séquentielle d'un programme parallèle se ramène entièrement à la vérification d'un certain nombre de prédicats concernant la seule version séquentielle (ou plus exactement la version monoprocessus ordonnée) – donc, finalement, à des vérifications de certaines propriétés *d'un programme séquentiel*. Nous nous attendons donc à ne pas nous trouver en butte aux problèmes de complexité auxquels nous avons fait allusion (chap.1) concernant la vérification empirique de propriétés de linéarisabilité ou de consistance séquentielle [17, 20]. Dans la même veine, concernant le non blocage en monoprocessus, une hypothèse de notre théorème, la vérification de cette condition se ramène à la vérification de la formule de non blocage précédemment donnée (§4.4), formule qui se réfère, elle aussi, à la sémantique séquentielle. Si complexe, voire infaisable, que cette vérification puisse se révéler suivant les cas, remarquons du moins que le problème n'est pas de même nature que s'il s'agissait de vérifier le non blocage dans des exécutions *parallèles*. (Sur la vérification de non blocage dans des formes *différentes* de parallélisme, on pourra se reporter par exemple à [28] [1].)

6.3 L'idée d'exécution séquentielle "test"

Il est fréquent qu'un certain programme parallèle soit destiné à être exécuté de nombreuses fois, sur des *données* différentes. Ces données sont des *paramètres* dans notre sens, comme nous l'avons évoqué (§2.1). Il est intéressant d'envisager les situations – assez fréquentes en pratique – où il peut être établi statiquement, sur le texte du programme, que *la propriété d'équivalence sémantique qui nous intéresse ne dépend pas des valeurs de*

1. Dans [28], l'auteur étudie le problème de la détection statique de blocages dans des programmes constitués de tâches définies statiquement, destinées à s'exécuter en parallèle, avec des synchronisations entre elles. Considérant une approximation conservatrice de ce problème, il développe un algorithme résolvant cette approximation en temps polynomial, puis montre expérimentalement l'intérêt de cette approximation.

ces paramètres. Intuitivement, il s'agit de situations où se présente un certain *découplage entre les données et le contrôle*, tel que les "variables de contrôle" ne dépendent pas des données. Ces situations correspondent à une certaine généralisation de la notion de *programme à contrôle statique* ([13], par exemple)[2].

Dès lors que sera établie cette propriété d'indépendance de l'équivalence sémantique par rapport aux valeurs des paramètres, il sera pertinent de considérer une instance du programme, obtenue en donnant des valeurs aux paramètres, et d'entreprendre de vérifier la correction du programme par *l'observation d'une exécution séquentielle* de cette instance – une démarche de vérification dynamique qui n'aurait pas de sens si elle ne devait apporter d'enseignements que sur cette seule instance.

Nous allons donner des conditions suffisantes pour que la correction d'un programme parallèle ne dépende pas des paramètres ; puis nous allons évoquer des applications possibles. Enfin, nous esquisserons à grands traits les principes d'une vérification dynamique de correction sur une exécution séquentielle.

Sur l'indépendance de la correction par rapport aux données

Nous considérons donc un programme parallèle écrit dans le langage que nous avons défini. Nous allons démontrer un théorème établissant, sous certaines hypothèses, l'indépendance de la correction (dans notre sens) par rapport aux valeurs des paramètres.

Théorème 2 *On suppose qu'il existe un ensemble Δ de variables et de paramètres du programme, vérifiant les hypothèses suivantes :*

i. Tous les paramètres sont dans Δ ;

ii. Dès lors qu'une référence à un élément de Δ figure en entrée d'une instruction, les références en sortie de cette instruction désignent toutes des variables de Δ ;

iii. Aucune référence à un élément de Δ ne figure dans une borne de boucle DO *ou* PDO*, ni dans un test de* IF *ou de* WHILE *;*

2. Dans [13], un programme est à contrôle statique si les bornes de ses boucles dépendent seulement de constantes numériques, d'indices de boucles englobantes et de *paramètres de structure*, un ensemble de variables entières définies une seule fois dans le programme, soit comme paramètres dans notre sens, soit à partir d'autres paramètres de structure définis antérieurement. Nous nous plaçons dans une situation différente, dans la mesure où "nos" variables de contrôle peuvent être modifiées plusieurs fois au cours de l'exécution du programme, au point de ne pas être connaissables statiquement – nous ne nous plaçons donc pas dans une situation de *contrôle statique* à proprement parler, mais plutôt dans une situation de *contrôle indépendant des données*.

iv. Aucune référence à un élément de Δ ne figure dans une expression d'indice de tableau.

Alors la vérification des hypothèses du théorème 1 ne dépend pas des valeurs des paramètres. Par conséquent, si une instance de programme vérifie les hypothèses du théorème 1, toute autre instance du programme dont la version séquentielle ne produit pas d'erreur d'exécution est sémantiquement équivalente à sa version séquentielle.

Preuve : Il s'agit de démontrer que, sous les hypothèses que nous venons de donner, celles du théorème 1 ne dépendent pas des valeurs des paramètres. Montrons d'abord que Δ contient toutes les variables dont les valeurs dépendent de celles des paramètres *(fermeture causale de Δ)*. Au cours de l'exécution séquentielle d'une instance du programme, soit x une variable, non élément de Δ, écrite par une instance d'instruction α. Il y aurait deux manières, pour la valeur de x ainsi écrite, de dépendre directement de variables éléments de Δ : soit de telles variables figureraient en entrée de α, circonstance exclue par *(ii)* ; soit la condition d'exécution de α, $\mathrm{Exe}^{\mathsf{s}}(\alpha)$, contiendrait des variables éléments de Δ, circonstance exclue par *(iii)* (§3.1).

C'est ainsi que, par ailleurs. le prédicat $\mathrm{Exe}^{\mathsf{s}}$ ne dépend pas des paramètres, et la terminaison du programme non plus. Selon *(iv)*, le prédicat Dep ne dépend pas non plus des paramètres (§3.2). Nous savons que Pre^0 est indépendant de toute variable (§3.3.1). Selon *(iv)*, les références à des événements sont indépendantes des paramètres ; donc, compte tenu de la non dépendance du prédicat $\mathrm{Exe}^{\mathsf{s}}$ par rapport aux paramètres (donc, de l'exécution des synchronisations dans la sémantique séquentielle), tout le mécanisme des synchronisations ne dépend pas des valeurs des paramètres : les hypothèses S1 et S2 (§3.3.3) n'en dépendent donc pas, non plus que la condition de non blocage (§4.4). ◀

Il importe de mentionner une faiblesse relative de cette démarche : le fait que les *tests* ne doivent pas dépendre des données est sans doute une limitation en pratique.

Quelques exemples

Un tel découplage entre les données et le contrôle se présentera assez souvent dans des programmes de calcul scientifique. Considérons par exemple des programmes mettant en œuvre des *méthodes d'éléments finis*. Nous voulons résoudre un système d'équations sur un certain domaine physique, par une approximation utilisant un *maillage* de ce domaine et calculant un ensemble discret de grandeurs associé à ce maillage – par exemple, des grandeurs aux nœuds de ce maillage. Il se produira généralement que les paramètres définissant le maillage (nombre de nœuds, contiguïtés...) soient les

paramètres de contrôle du programme, influençant les nombres d'itérations de boucles et intervenant dans les indices des vecteurs et des matrices ; tandis que les grandeurs physiques aux nœuds du maillage – températures, pressions, vitesses, proportions d'espèces chimiques... – constitueront des "variables dépendant des données", éléments de Δ dans notre terminologie : des variables n'agissant pas sur la structure de contrôle du programme.

Dès lors, considérant un programme parallèle d'éléments finis qui présente un tel découplage, la correction séquentielle de ce programme ne dépendra pas (dans le cas, par exemple, d'un problème d'évolution temporelle) des conditions initiales ni des conditions aux limites, paramètres de données sur lesquels on fera tourner le programme, mais seulement de la structure de contrôle du programme, correspondant au maillage.

On peut même concevoir un certain programme parallèle d'éléments finis permettant une déformation (continue) du maillage, pour ajustement à des domaines physiques de géométries différentes. Sous la condition que les paramètres de déformation et d'ajustement constituent des "variables dépendant des données" dans notre sens, la correction séquentielle du programme parallèle sera un invariant de la déformation en question.

En revanche, bien entendu, un *changement des équations*, en vue de traiter un problème de physique différent avec le même maillage, modifie les relations de dépendance entre variables, et ne conserve donc pas la correction séquentielle.

La vérification dynamique d'une instance de programme

A la lumière des considérations qui précèdent, il peut donc être pertinent, dans un certain nombre de cas, de vouloir vérifier *dynamiquement* la correction d'une instance de programme, par l'observation de l'exécution de sa version séquentielle (plus exactement, de sa version monoprocessus ordonnée).

Soit une instance de programme parallèle dont il est question de vérifier la correction séquentielle dans notre sens. *Nous supposons savoir, par un moyen quelconque, que sa version monoprocessus ordonnée se termine (c'est-à-dire ne boucle pas indéfiniment).* Sous cette réserve, il est simple d'observer que *le problème de la correction séquentielle de notre instance de programme est décidable*, par une procédure qui consiste à exécuter la version monoprocessus ordonnée (un programme séquentiel, donc), en *observant*, "de l'extérieur", l'exécution en train de se dérouler, afin de vérifier les hypothèses du théorème 1 (autres que la terminaison, supposée ici).

Dans le cas d'un programme vérifiant les hypothèses du théorème 2 et la condition de terminaison – le cas où il peut être pertinent d'avoir recours à

une telle vérification dynamique – cette démarche peut, à grands traits, se décrire comme suit.

Tout d'abord, nous tentons une analyse statique du texte du programme, afin de tenter d'y vérifier les hypothèses du théorème 1. Nous répertorions celles de ces hypothèses que nous ne parvenons pas à vérifier statiquement – par exemple, des dépendances possibles qui ne paraissent pas être préservées par des précédences, une incertitude sur la condition de non blocage, ou des relations $\mathrm{Pre}^{\mathsf{s}}$ entre synchronisations, exigées par les hypothèses S1 et(ou) S2, que l'on ne parvient pas à vérifier.

Nous considérons alors un jeu de valeurs des paramètres "valide", c'est-à-dire ne produisant pas d'erreur d'exécution en séquentiel, et nous lançons l'exécution monoprocessus ordonnée de l'instance correspondante. Ou bien plutôt, si nous le pouvons, nous préférerons considérer une *exécution monoprocessus à paramètres muets*, c'est-à-dire une exécution au cours de laquelle, dans la succession des instances, les calculs des variables de Δ ne sont pas effectués – ce qui peut représenter un gain de temps. C'est possible en principe, en raison des hypothèses du théorème 2.

Soit par exemple l'instruction :

```
A(I,J+1) = B(I,J-N(I))*N(I-1)
```

où `A()` et `B()` sont des tableaux de variables éléments de Δ, `N()` est un tableau de variables "de contrôle", non éléments de Δ, et `I` et `J` sont des indices de boucles englobantes. Si par exemple l'instance `I=2,J=3` de cette instruction est exécutée, il est procédé à une *évaluation partielle*, des seules expressions de contrôle. D'abord, les valeurs des indices de boucles sont considérées, ce qui conduit à instancier l'instruction :

```
A(2,3+1) = B(2,3-N(2))*N(2-1)
```

Puis, les variables de contrôle sont évaluées – supposons par exemple que l'on ait, à l'exécution de cette instance : `N(1)=3` et `N(2)=6` ; effectuant les expressions d'indices, nous obtenons :

```
A(2,4) = B(2,-3)*3
```

Alors, nous évitons d'évaluer les variables de Δ, et nous prenons note que cette instance d'instruction fait intervenir :

- en sortie : `A(2,4)`
- en entrée : `B(2,-3), N(2), N(1)`

Au cours de cette exécution en monoprocessus, il est procédé aux observations suivantes :

- Une situation de blocage sur un WAIT est observable – l'exécution monoprocessus ordonnée s'arrête tout simplement à ce point. (Si l'on observe plutôt la version séquentielle, dans laquelle les synchronisations sont "désactivées" mais examinées tout de même, il est aisé de vérifier, sur chaque WAIT rencontré, si l'événement correspondant a été "posté".)

- Au cours de l'exécution, il est pris note des références d'événements apparaissant successivement, ce qui permet notamment de répertorier les paires de synchronisations $\mathrm{Sync}^{\mathsf{s}}$, ainsi que les paires d'instances de synchronisations entre lesquelles les hypothèses S1 et S2 prescriraient une précédence $\mathrm{Pre}^{\mathsf{s}}$ non établie lors de l'analyse statique. La recherche de ces paires d'instances, faisant appel à des comparaisons binaires entre références, est de complexité polynomiale en le nombre d'instances ainsi examinées.

- Au cours de l'exécution des constructions parallèles ayant posé des problèmes lors de l'analyse statique, il est pris note des références de variables (autres que les événements), comme nous l'avons vu sur l'exemple ci-dessus, de manière à répertorier les relations de dépendance Dep dont la préservation n'a pas pu être établie lors de l'analyse statique. Plus précisément, l'analyse statique ayant repéré toutes les paires d'instructions (a, b) entre lesquelles risquait d'exister une dépendance non préservée, il est pris note des paires (α, β) d'instances respectivement de a et b entre lesquelles apparaît effectivement une dépendance $\mathrm{Dep}(\alpha, \beta)$. La recherche de ces paires (α, β), faisant appel à des comparaisons binaires entre références, est de complexité polynomiale en le nombre d'instances des instructions des paires (a, b) à considérer.

Cela fait, nous sommes en présence de deux problèmes de graphes :

- vérifier si les liaisons Dep répertoriées font partie de la fermeture transitive du graphe constitué par les liaisons $\mathrm{Sync}^{\mathsf{s}}$ répertoriées et par les liaisons Pre^{0} (représentables facilement) (avec la restriction, liée à notre définition de $\mathrm{Pre}^{\mathsf{s}}$ (§3.3.2), que les chemins envisagés, dans le graphe des Pre^{0} et $\mathrm{Sync}^{\mathsf{s}}$, ne se terminent pas par des $\mathrm{Sync}^{\mathsf{s}}$) ;

- considérant les paires d'instances de synchronisations (θ, γ) entre lesquelles il faut vérifier une précédence $\mathrm{Pre}^{\mathsf{s}}$, regarder si, soit la liaison de θ vers γ, soit la liaison de γ vers θ, est dans la même fermeture transitive considérée ci-dessus, avec la même restriction.

Il s'agit de problèmes de graphes assez connus, faisant appel à des algorithmes de parcours de graphes *en largeur d'abord*. On se reportera par exemple à [8, 38]. Ces algorithmes de calcul de fermeture transitive sont de complexité polynomiale en le nombre de nœuds du graphe, qui est, au plus, le nombre d'instances d'instructions dans les constructions parallèles n'ayant pas pu être vérifiées lors de l'analyse statique pour le premier problème, le nombre d'instances de synchronisations pour le deuxième problème.

La procédure de vérification dynamique, esquissée ici, a une complexité algorithmique qui est, dans le pire des cas, polynomiale en le nombre d'*instances* d'instructions exécutées – donc "grande" tout de même. D'où l'importance, si l'on envisage l'utilisation d'une telle procédure, d'épuiser en premier lieu toutes les possibilités de l'analyse statique (comme nous l'avons mentionné), afin de "cibler" la vérification dynamique ; et de réserver une telle procédure à l'examen de programmes destinés à être utilisés très fréquemment.

6.4 Le traitement incrémental des non-préservations de dépendances

Soit un programme parallèle sur lequel nous tentons de vérifier les hypothèses du théorème 1 (par analyse statique ou vérification dynamique). Examinons ce qui peut advenir lorsqu'il s'avère qu'une dépendance *n'est pas* (ou pas nécessairement) préservée par une précédence. Nous avons vu qu'il s'agit de vérifier des relations de la forme :

$$\mathrm{Exe}^{\mathsf{s}}(\alpha) \wedge \mathrm{Exe}^{\mathsf{s}}(\beta) \;\&\; \mathrm{Dep}(\alpha,\beta) \Rightarrow \mathrm{Pre}^{\mathsf{s}}(\alpha,\beta)$$

(où α et β sont des instances d'instructions), dont les prédicats font référence, nous l'avons vu, à la seule sémantique séquentielle.

Supposons que, dans une instance de programme parallèle que nous nous donnons, une relation de dépendance ne soit pas préservée – qu'il existe deux instances d'instructions α et β, exécutées et en dépendance $\mathrm{Dep}(\alpha,\beta)$, pour lesquelles l'implication ci-dessus ne soit pas vérifiée. Il peut alors se faire que, dans une certaine exécution parallèle, ces instances ne s'exécutent pas dans le bon ordre, de sorte que la variable concernée par la dépendance $\mathrm{Dep}(\alpha,\beta)$ ne reçoive pas sa valeur "séquentielle". Dans cette exécution, par conséquent, la valeur "erronée" pourra propager ses effets, non seulement dans les calculs ultérieurs d'autres variables faisant intervenir cette valeur comme opérande, mais aussi (en raison de la référence dynamique) dans des *désignations* de variables auxquelles s'appliqueront de tels calculs, et (en raison de l'intervention de variables dans les bornes de boucles et les tests des IF et des WHILE) dans les conditions d'exécution des instances d'instructions.

Enfin, bien entendu, cette propagation d'erreur en cascade pourra créer des erreurs d'exécution.

Cependant, bien que la non préservation d'une dépendance puisse ainsi avoir des conséquences en cascade, potentiellement ravageuses, sur toute la suite de l'exécution du programme parallèle, il peut être instructif d'observer que, moyennant quelques précautions, la *réparation* par le programmeur de cette non préservation – par l'adjonction de synchronisations adéquates, par exemple – *ne remet pas en cause* les vérifications déjà faites : les dépendances qui étaient préservées avant la réparation le demeurent après. Il y a un caractère **incrémental** de la procédure de vérification et de réparation.

Ce caractère incrémental est une conséquence de notre constante référence à la sémantique séquentielle.

Il y a des précautions à prendre pour assurer ce caractère incrémental. En présence d'une dépendance non préservée, l'idée est de *renforcer* la précédence $\mathrm{Pre}^{\mathsf{s}}$ de telle sorte que la dépendance en question devienne préservée par la précédence ainsi renforcée. Il doit bien s'agir d'un *renforcement*, en ce sens que l'on doit avoir, sur l'ensemble du programme, une implication $\mathrm{Pre}^{\mathsf{s}}_{avant} \Rightarrow \mathrm{Pre}^{\mathsf{s}}_{après}$ où $\mathrm{Pre}^{\mathsf{s}}_{avant}$ et $\mathrm{Pre}^{\mathsf{s}}_{après}$ désignent respectivement les précédences $\mathrm{Pre}^{\mathsf{s}}$ avant et après la réparation. C'est ainsi que les implications $\mathrm{Dep} \Rightarrow \mathrm{Pre}^{\mathsf{s}}$ antérieurement vérifiées, le demeurent après.

Si ce renforcement de $\mathrm{Pre}^{\mathsf{s}}$ se fait par l'adjonction de nouvelles synchronisations ou le déplacement de synchronisations existantes, il faut veiller à ce que les hypothèses S1 et S2 concernant les synchronisations (§3.3.3) demeurent vérifiées après la réparation.

Examinons, sur un petit exemple, comment les choses peuvent se présenter. Sur la figure 6.1, est représenté un élément de programme. La version séquentielle (qui donne la référence de la sémantique souhaitée) est donnée à droite.

Sur cet élément de programme, nous supposons que les instructions non détaillées (représentées par "...") ne comportent pas de références aux variables $A()$ ni P, ni de synchronisations. Par ailleurs, nous supposons que cet élément de programme n'est pas lui-même inclus dans une construction parallèle plus vaste.

Le lecteur attentif se rend aisément compte que cet élément de programme est *incorrect* dans notre sens, présentant des conflits mémoire portant sur la variable P : des dépendances de `c1` et `c2` d'une part, vers `d1`, `a` et `p` d'autre part, ne sont pas préservées. Il faudra corriger cela, mais, en vertu de ce que nous avons remarqué sur le caractère incrémental de la vérification, cela ne nous interdit nullement d'examiner si les dépendances portant sur les variables $A()$, elles, sont préservées.

Intéressons-nous donc à la dépendance entre les instructions `b` et `a`, associée au tableau de variables $A()$. Nous remarquons que cette dépendance, ainsi que les précédences que nous allons rechercher, dépendent de la variable

```
      B=...                                B=...
      ...                                  ...
      psections
        section
          ...                              ...
          if(B) then                       if(B) then
c1:         P=2                              P=2
          else                             else
c2:         P=3                              P=3
          endif                            endif
f:        ...                              ...
        section
d1:       do J=1,P
q:          post E(J)
          enddo
          ...                              ...
d2:       pdo I=1,N                        do I=1,N
            ...                              ...
w:          wait E(I)
b:          ...=A(I)                         ...=A(I)
a:          A(I+P)=...                       A(I+P)=...
p:          post E(I+P)
          endpdo                           enddo
      endpsections
      ...                                  ...
```

FIG. 6.1: *Un exemple de vérification. A droite, la version séquentielle.*

P. Comme c'est la version séquentielle qui nous sert de référence sémantique ici (en vertu de notre théorème), nous n'avons pas à nous préoccuper pour l'instant des conflits mémoire ci-dessus remarqués concernant P, et nous considérons la valeur "séquentielle" de P (égale à 2 ou 3) en `b` et `a`. Nous nous occupons donc de la dépendance de `a` vers `b`, associée à la variable $A()$. P étant positif, c'est cette dépendance qui devra être préservée afin que les valeurs de $A()$ calculées par `a` soient, le cas échéant, celles utilisées par `b`, P itérations plus tard.

En choisissant d'incorporer, dans le prédicat Dep, les conditions $\text{Exe}^{\mathtt{s}}$ d'exécution, nous obtenons :

$$\text{Dep}(a_x, b_y) = (1 \leq x \leq N) \wedge (1 \leq y \leq N) \wedge (y > x) \wedge (y = x + [\![P]\!]@a_x)$$

Expliquons cette relation. a_x (resp. b_y) désigne l'instance de l'instruction `a` (resp. `b`) correspondant à l'itération x (resp. y). Des quatre termes de cette conjonction, le premier désigne la condition d'exécution de `a`, le deuxième la condition d'exécution de `b`, le troisième désigne la condition que a_x soit exécuté avant b_y dans la version séquentielle, et le quatrième exprime que les deux instances a_x et b_y accèdent à la même variable (à la même case du tableau $A()$), dans la version séquentielle. Comme précédemment, $[\![P]\!]@a_x$ désigne la valeur de P lue par l'instance a_x dans la version séquentielle[3].

Recherchons maintenant les précédences qui peuvent exister de `a` vers `b`. Il est aisément détectable que la précédence de contrôle Pre^0 ne suffit pas, et qu'il faut faire intervenir les synchronisations $\text{Sync}^{\mathtt{s}}$. Plus précisément, considérons un chemin :

$$a_x \rightarrow p_u \rightsquigarrow w_z \rightarrow b_y$$

(Nous utilisons les notations introduites en §3.3.2). Il s'agit ici de calculer si nous avons une précédence utilisant une liaison de synchronisation. Exprimons l'éventuelle relation de précédence :

$$\text{Pre}^{\mathtt{s}}(a_x, b_y) = \text{Pre}^0(a_x, p_u) \wedge \text{Exe}^{\mathtt{s}}(p_u) \wedge \text{Sync}^{\mathtt{s}}(p_u, w_z) \wedge \text{Exe}^{\mathtt{s}}(w_z) \wedge \text{Pre}^0(w_z, b_y)$$

En ce qui concerne la synchronisation $\text{Sync}^{\mathtt{s}}$, nous supposons pouvoir vérifier ici qu'il y a bien synchronisation de `p` vers `w`, en ce sens que les événements $E()$ ne se trouvent pas déjà *postés* à l'entrée de l'élément de programme considéré ici.

Nous avons donc, en exprimant successivement les cinq termes de la conjonction :

$$\text{Pre}^{\mathtt{s}}(a_x, b_y) = (x = u) \wedge (1 \leq u \leq N) \wedge (u + [\![P]\!]@p_u = z) \wedge (1 \leq z \leq N) \wedge (z = y)$$

3. Pour alléger nos écritures, nous avons supposé que N est un paramètre. Dans le cas où il s'agit d'une variable, dans les formules données ici, "N" devrait être remplacé par "$[N]@d2$".

Il s'agit de vérifier s'il existe un tel chemin de précédence tel que $\mathrm{Dep}(a_x, b_y) \Rightarrow \mathrm{Pre}^{\mathtt{s}}(a_x, b_y)$, c'est-à-dire s'il existe une instance p_u de `p` et une instance w_z de `w` tels que l'on ait cette implication. Dans notre expression de $\mathrm{Pre}^{\mathtt{s}}$, nous tentons d'éliminer les inconnues u et z :[4]

– Eliminer u par $(x = u)$:

$$\mathrm{Pre}^{\mathtt{s}}(a_x, b_y) = (1 \leq x \leq N) \wedge (x + [\![P]\!]@p_x = z) \wedge (1 \leq z \leq N) \wedge (z = y)$$

– Eliminer z par $(z = y)$:

$$\mathrm{Pre}^{\mathtt{s}}(a_x, b_y) = (1 \leq x \leq N) \wedge (x + [\![P]\!]@p_x = y) \wedge (1 \leq y \leq N)$$

à rapprocher de :

$$\mathrm{Dep}(a_x, b_y) = (1 \leq x \leq N) \wedge (1 \leq y \leq N) \wedge (y > x) \wedge (y = x + [\![P]\!]@a_x)$$

Nous avons bien $\mathrm{Dep}(a_x, b_y) \Rightarrow \mathrm{Pre}^{\mathtt{s}}(a_x, b_y)$, moyennant la condition $[\![P]\!]@p_x = [\![P]\!]@a_x$, qui est assurée dès lors que nous vérifions que P n'est pas réécrit entre les deux lectures *sur la version séquentielle*, ce que nous avons supposé.

Remarquons que l'expression $\mathrm{Dep}(a_x, b_y)$ implique ici que $[\![P]\!]@a_x > 0$, ce qui doit être le cas pour que l'on ait effectivement la dépendance envisagée, et qui est bien vérifié sur notre exemple.

Nous devons nous occuper maintenant de la variable P. Nous avons déjà mentionné que les dépendances, associées à P, de `c1` et `c2` d'une part, vers `d1`, `a` et `p` d'autre part, ne sont pas préservées. Une réparation radicale consiste à "reséquentialiser" le PSECTIONS – mais cela peut présenter un coût important lorsque les instructions `f` sont longues à exécuter. Une modification plus fine consiste à introduire une synchronisation, de la fin du calcul de P vers le début de son utilisation. Cette synchronisation ne peut utiliser un événement $E()$ déjà en usage dans les boucles `d1` et `d2`. Une possibilité est d'utiliser un autre événement (ici F), dûment réinitialisé avant et après usage (pour le cas où il serait déjà utilisé ailleurs dans le programme). Moyennant cette précaution, toutes les préservations de dépendances déjà vérifiées (dont celles concernant $A()$ ici) demeurent. Une rectification possible de notre élément de programme est montrée ici (figure 6.2).

4. Ces principes de traitement algorithmique des dépendances et des précédences ont été illustrés dans nos applications [36, 4], et [5] reproduit en annexe C.

```
      B=...
      ...
      CLEAR F                          ajout
      psections
        section
          ...
          if(B) then
c1:         P=2
          else
c2:         P=3
          endif
          POST F                       ajout
f:        ...
        section
          WAIT F                       ajout
d1:       do J=1,P
q:          post E(J)
          enddo
          ...
d2:       pdo I=1,N
            ...
w:          wait E(I)
b:          ...=A(I)
a:          A(I+P)=...
p:          post E(I+P)
          endpdo
      endpsections
      CLEAR F                          ajout
      ...
```

FIG. 6.2: *L'exemple précédent, corrigé*

Chapitre 7

Extensions possibles

Dans l'étude que nous avons développée, nous n'avons pas traité de toutes les constructions parallèles qui ont été proposées. Nous allons esquisser ici des extensions possibles. Tout d'abord, nous allons évoquer les *synchronisations à passage de références*. Puis, sortant du cadre que nous nous sommes fixés jusqu'à ce point, dans lequel était recherchée l'équivalence sémantique entre un programme parallèle et sa version séquentielle, nous allons traiter des *sections critiques*, une construction parallèle très utilisée. En cette occasion, nous allons évoquer la possibilité de s'intéresser à des propriétés de similitude entre programme parallèle et programme séquentiel, plus faibles (c'est-à-dire moins exigeantes) que l'équivalence sémantique que nous avons considérée.

7.1 Les synchronisations à passage de références

Dans notre modèle d'exécution (§2.2), nous avons vu que, lorsqu'un POST ou un WAIT est exécuté dans une construction parallèle, il est réalisé une mise à jour des variables partagées dans cette construction. On pourrait envisager, dans une perspective d'économie, de limiter cette mise à jour à une ou plusieurs référence(s) spécifiée(s) dans l'instruction POST ou WAIT. La syntaxe des synchronisations, ainsi enrichie, se présenterait ainsi :

```
POST (<evenement>, <liste_references>)
WAIT (<evenement>, <liste_references>)
CLEAR (<evenement>)
```

Les instructions POST et WAIT, ainsi enrichies, comporteraient en argument une liste de références, dont le premier élément est la référence, dans l'environnement d'exécution, de l'événement – qui fonctionne de la même manière que précédemment – tandis que les autres éléments sont les références, dans l'environnement d'exécution, des variables sur lesquelles porte la mise à jour. La figure 7.1 présente un petit exemple (inspiré directement de celui de la figure 2.1).

```
a0:     A(0)=...
p0:     post (E(0),A(0))
        pdo I=1,N
          ...
w:        wait (E(I-1),A(I-1))
b:        ...=A(I-1)
a:        A(I)=...
p:        post (E(I),A(I))
          ...
        endpdo
```

FIG. 7.1: *Un exemple d'utilisation du post/wait à passage de références*

Sur cet exemple, la synchronisation assurerait, comme précédemment, que la valeur de $A(I-1)$ utilisée par l'instruction **b** à l'itération I est la valeur calculée par l'instruction **a** à l'itération $I-1$ (ou, pour $I=1$, par l'instruction **a0**) ; mais cette fois, cette précédence ne porterait pas sur d'éventuelles autres variables présentes et ne figurant pas ici. Il serait très possible d'introduire dans notre formalisme – moyennant cependant une certaine lourdeur supplémentaire – de telles "synchronisations limitées" ; cela supposerait que nos prédicats $\mathrm{Sync}^{\mathtt{s}}$ et $\mathrm{Pre}^{\mathtt{s}}$ comportent comme arguments, non plus seulement les instances de "départ" et d'"arrivée", mais les références pour lesquelles les précédences correspondantes sont assurées. Les conséquences de ces précédences en ce qui concerne les dates d'exécution (§4.3) seraient inchangées. Par conséquent, nos démonstrations pourraient aisément être étendues pour inclure ces synchronisations à passage de références.

7.2 Les sections critiques

Un type de construction parallèle que nous n'avons pas introduit dans notre langage (chapitre 2), et qu'il est intéressant de considérer ici, est constitué par les *sections critiques*. Considérons l'exemple simple de la figure 7.2.

La boucle séquentielle à gauche de la figure réalise le calcul des $A(I)$, et fournit leur somme, dans T. $A()$ est ici d'un type numérique (entier ou réel), le même que T. Nous supposons, dans cet exemple, que les calculs des $A(I)$ sont indépendants, et peuvent donc être effectués en parallèle. Pour le calcul de cette somme T, l'ordre dans lequel les $A(I)$ sont additionnés n'importe pas[1]. Il peut donc être intéressant de faire en sorte que l'ordre d'exécution des instances de l'instruction **s** ne soit pas imposé ; il faut cepen-

1. Ici et dans la suite, nous *négligeons* les problèmes soulevés par l'approximation numérique en machine, en conséquence desquels, par exemple, l'addition numérique, telle que réalisée en machine, n'est pas nécessairement commutative, ou associative.

```
        T=0                          T=0
        do I=1,N                     pdo I=1,N
          ...                          ...
          A(I)=...                     A(I)=...
                                       critical section(V,T)
s:        T=T+A(I)                       T=T+A(I)
                                       end critical section(V,T)
          ...                          ...
        enddo                        endpdo
```

FIG. 7.2: *Un exemple simple de section critique. A gauche, la version séquentielle.*

dant assurer que ces instances *ne s'exécutent pas en même temps*, et que les valeurs intermédiaires obtenues pour T sont transmises entre ces instances successives. C'est le rôle de la *section critique* que nous faisons figurer dans la parallélisation (à droite de la figure). La variable V figurant dans l'instruction `critical section` est un *verrou*, qui peut prendre les valeurs *locked* ou *unlocked*. Un même verrou V devra être partagé par toutes les instances de sections critiques qui, précisément, ne doivent pas s'exécuter en même temps. Les variables mentionnées après V dans la liste d'arguments de l'instruction `critical section` – dans l'exemple, la seule variable T – sont les variables sur lesquelles portera la mise à jour (nous introduisons d'emblée un mécanisme de "passage de références" analogue à celui qui vient d'être décrit (§7.1) pour les synchronisations).

Afin d'assurer le comportement souhaité, le modèle d'exécution des sections critiques est conçu ainsi.

- Au départ, la valeur de V est *unlocked*.
- Lorsqu'un processus rencontre une instruction CRITICAL SECTION, il examine la valeur du verrou. Si cette valeur est *locked*, il attend et recommence plus tard. Si cette valeur est *unlocked*, il donne au verrou la valeur *locked*, et poursuit l'exécution ("entrant" ainsi dans la section critique), après avoir mis à jour les variables référencées après V dans la liste d'arguments.
- Lorsqu'un processus rencontre une instruction END CRITICAL SECTION, il met à jour les variables référencées après V dans la liste d'arguments, puis donne au verrou (qui a à ce moment-là la valeur *locked*) la valeur *unlocked*, puis poursuit l'exécution ("sortant" ainsi de la section critique).

Introduire ici les sections critiques, c'est *sortir du cadre dans lequel nous*

nous sommes situés jusqu'à présent, où était souhaitée l'équivalence sémantique entre un programme parallèle et sa version séquentielle. En effet, dans l'exemple simple ci-dessus, l'idée est de *ne pas exiger que les valeurs intermédiaires reçues par T dans le courant de l'exécution soient identiques à celles obtenues dans la version séquentielle.* Nous demandons seulement que la valeur *finale* de T au sortir de cette construction parallèle, soit la même dans les deux versions.

Il s'agit de rechercher comment les précédents résultats peuvent être étendus, modifiés, afin de permettre l'introduction des sections critiques dans les programmes envisagés ici.

Sans pouvoir prétendre traiter les sections critiques dans toute leur généralité ici, nous allons cependant nous pencher sur un schéma déjà relativement général, présenté dans la figure 7.3. L'élément de programme ici représenté peut lui-même être inclus dans une construction parallèle plus vaste.

Sur cette forme, $A()$ et T ne sont pas nécessairement d'un type numérique (entier ou réel), et peuvent correspondre à des structures beaucoup plus complexes. Nous nous cantonnons ici au cas où l'appel à la fonction *fonc()* se conforme à nos restrictions concernant l'appel à des sous-programmes (§2.1). C'est la variable T qui, figurant à la fois en lecture et en écriture dans la section critique, va nous intéresser ici. (Une telle variable est dite *variable de réduction.*) Il serait facile d'étendre notre schéma à une situation où plusieurs instructions figurent dans la section critique, à condition, toutefois, soit d'étendre à toutes les variables y figurant à la fois en lecture et en écriture ce que nous allons dire concernant T, soit de "condenser" ces instructions sous une forme équivalente du point de vue sémantique (lectures-écritures de variables) à un appel à une fonction, selon la forme ci-dessous, ce qui peut conduire à "condenser" dans la variable de réduction T, en lui donnant un type suffisamment complexe, *toutes les informations qui passeront d'une instance à une autre de cette section critique.*

L'utilisation de ce schéma de section critique peut présenter un grand intérêt dans la situation où le calcul de $A(I)$ par `a` est coûteux en temps, tandis que l'appel à *fonc* est relativement court. Dans une telle situation, en effet, les longs calculs des $A(I)$ peuvent être effectués plusieurs à la fois, en parallèle, tandis que les N appels à *fonc* doivent être effectués séparément.

Avant de donner les conditions de validité de cette parallélisation, illustrons les possibilités assez étendues qu'offre ce schéma en évoquant un exemple : celui de l'*optimisation sous contraintes.* Chaque itération recherche, en parallèle, une solution admissible à un certain problème de contrainte, assortie d'un *coût.* A l'itération I, $A(I)$ reçoit pour valeur, soit cette solution admissible avec son coût, soit une valeur notée $\perp$ lorsque l'itération n'a pas trouvé de solution. T, de même type que $A(I)$, est initialisé à $\perp$. *fonc*(,) reçoit en entrée deux valeurs de ce même type, et retourne en sortie celle

```
          T=eps
          ...
          pdo I=1,N
            ...
a:          A(I)=...
            critical section(V,T)
s:            T=fonc(A(I),T)
            end critical section(V,T)
            ...
          endpdo
```

FIG. 7.3: *Un schéma de section critique*

des deux valeurs qui est "minimale" pour l'ordre suivant : l'ordre des coûts associés, complété par la donnée que $\perp$ est "plus grand" que les autres valeurs. En fin de boucle, T contient $\perp$ lorsque aucune solution admissible n'a été trouvée, et une solution de coût minimal dans le cas contraire. Dans une telle situation, et sous la seule réserve qu'il n'y ait pas *plusieurs* solutions de même coût minimal (réserve sur laquelle nous reviendrons plus loin), cette boucle parallèle retournera la même solution T que sa version séquentielle.

Dans ce schéma de section critique, il est aisé de voir que la parallélisation se justifie sous les deux conditions suivantes :

- Toutes les dépendances existant entre itérations de cette boucle, ou entre elles et le reste du programme, sont préservées par des précédences, à la seule exception d'une dépendance $\mathrm{Dep}(s, s)$ portant sur la variable `T`. (dépendance qui a pour expression : $\mathrm{Dep}(s_x, s_y) = (x < y)$)

  Par conséquent, dans cette boucle, `T` n'est ni lue, ni écrite, ailleurs que dans l'instruction `s`.

- La fonction *fonc()* a une propriété de *symétrie itérative*, ce qui signifie que le résultat final T de l'application de *fonc()* aux N valeurs $A(I)$ (T étant initialisé à *eps*), ne dépend pas de l'ordre dans lequel ces N valeurs sont traitées.

En effet, moyennant la première condition, les valeurs intermédiaires de T, qui ne sont certes pas les mêmes dans une exécution parallèle **P** et dans l'exécution séquentielle **S**, ne sont pas utilisées ailleurs que dans les instances successivement exécutées de s. La section critique assure que, dans **P**, les instances de s s'exécutent à des dates d'exécution différentes et que les valeurs intermédiaires de T sont dûment transmises entre ces instances. Enfin, en vertu de la deuxième condition, ces calculs différents de T donnent la même valeur à la sortie de la boucle.

Affaiblir l'exigence d'équivalence sémantique

Dans le problème d'optimisation sous contraintes que nous avons considéré à titre d'exemple, nous avons mentionné la réserve qu'il n'existe pas plusieurs solutions de même coût minimal. Lorsque cette réserve est levée, notre fonction *fonc()* n'a plus la propriété de symétrie itérative ; cette observation simple suggère des possibilités d'affaiblir l'exigence d'équivalence sémantique.

Dans les chapitres précédents, nous nous sommes placés dans la situation où un programme parallèle devait être sémantiquement équivalent à sa version séquentielle, c'est-à-dire, où les diverses variables devaient connaître les mêmes calculs, donnant donc les mêmes résultats, dans l'exécution séquentielle **S** et dans toute exécution parallèle **P**. Nous venons d'esquisser, avec les sections critiques, l'étude d'une situation où cette exigence d'équivalence sémantique est affaiblie de la manière suivante : certains calculs intermédiaires n'ont pas à être identiques dans les deux exécutions, dès lors qu'une propriété mathématique de symétrie itérative permet de "retomber" sur des valeurs égales au terme de ces calculs intermédiaires.

On peut imaginer d'affaiblir encore davantage cette exigence d'équivalence sémantique, selon l'idée générale suivante : il s'agit toujours de prescrire que l'exécution séquentielle **S** et toute exécution parallèle **P** présentent *une même propriété quant aux calculs qu'elles produisent*, mais sans que cette propriété commune soit nécessairement l'équivalence sémantique que nous avons considérée précédemment, exigence en quelque sorte "maximale" que l'on puisse prescrire dans ce contexte.

Dans l'exemple précédemment évoqué, l'optimisation sous contraintes avec possibilité d'ex-aequo, nous obtenons une certaine équivalence affaiblie, qui peut s'exprimer ainsi : "lorsque **S** conclut à l'absence de solution admissible, **P** aussi ; lorsque **S** trouve une solution optimale, **P** trouve également une solution optimale, *mais pas nécessairement la même*." Il peut très bien se faire que cette équivalence plus faible convienne tout à fait au programmeur, dans une situation où la suite de son programme s'accommode bien de cette "équivalence" ainsi affaiblie. Il faudrait rechercher sous quelles conditions une démarche de "récurrence pas à pas le long d'une exécution parallèle", analogue à celle que nous avons suivie pour démontrer le théorème 1, permettrait de vérifier de telles propriétés d'"équivalence affaiblie".

Chapitre 8

Conclusion

Nous avons abordé le problème de la vérification de programmes parallèles dans un certain modèle de parallélisme : le modèle asynchrone à mémoire partagée. Ce modèle est largement utilisé en calcul scientifique et implémentable sur de nombreuses architectures parallèles.

Nous avons considéré un langage parallèle constitué par l'adjonction, à un langage impératif séquentiel très général, de quelques constructions parallèles : sections et boucles parallèles, synchronisations par événements. Nous nous intéressons à une propriété de **correction séquentielle** qui se définit comme une *équivalence sémantique* entre un programme parallèle, écrit dans ce langage, et sa *version séquentielle*, que nous définissons. Il est souhaité que le programme parallèle effectue les mêmes calculs, et produise donc les mêmes résultats, que sa version séquentielle, utilisant les ressources du parallélisme simplement pour être plus rapide.

Cette notion d'équivalence sémantique ici considérée n'est pas pertinente dans tout le domaine de la programmation parallèle (en toute généralité, il n'existe pas toujours une "version séquentielle" de référence dictant la sémantique souhaitée d'un programme parallèle), mais elle a semblé appropriée notamment dans le cas des programmes de calcul scientifique.

L'objet principal du présent travail était de présenter et de démontrer un théorème qui assure l'équivalence sémantique entre un programme parallèle et sa version séquentielle, sous un certain nombre d'hypothèses, principalement une condition de *préservation des dépendances de données*. Ces hypothèses portent seulement sur la sémantique de la version séquentielle : elles ne se réfèrent aucunement à des propriétés supposées d'une exécution parallèle particulière.

Cette référence à la seule sémantique séquentielle, qui est un aspect important de notre résultat, explique une certaine complexité de la démarche : en effet, aussi longtemps que l'on n'a pas prouvé, précisément, la correction de notre programme parallèle, on ne peut pas même supposer qu'une expression, évaluée au cours d'une exécution parallèle particulière, y a une valeur

bien déterminée. On ne peut pas davantage supposer que cette exécution parallèle est exempte d'erreur d'exécution – l'absence d'erreur d'exécution n'est ici supposée que pour la version séquentielle.

La démonstration de notre théorème utilise une notion de *date d'exécution*. Dans une exécution d'un programme parallèle, chaque exécution d'une instruction peut se voir attribuer une "date", de manière telle que les résultats d'un calcul effectué "à telle date" ne sont pas disponibles comme entrées avant "la date suivante". Ce schéma de date d'exécution peut être décrit comme une *discrétisation en temps préservant la causalité*. La partie principale de la démonstration utilise une récurrence sur la date d'exécution, afin d'établir que l'équivalence sémantique entre le programme parallèle considéré et sa version séquentielle, se propage le long de toute exécution parallèle. L'exigence de se référer à la sémantique *séquentielle* tandis que la récurrence procéde le long d'une exécution *parallèle*, explique les complications de la preuve.

Ce théorème a fait l'objet d'une application, au CERMICS, dans un outil de vérification de programmes parallèles écrits dans un certain sous-ensemble de notre langage [4, 5, 36]. Un travail de thèse actuellement en cours au CERMICS a pour objectif la formalisation de la sémantique des constructions parallèles dans un calcul de processus (le π-calcul) et sa représentation en *Coq*, un système d'assistant à la preuve [21, 22].

Ce théorème semble bien pouvoir se prêter à des applications plus étendues. Notamment, sa référence à la seule sémantique séquentielle permet d'appliquer, à la vérification de programmes parallèles, toutes les ressources de l'analyse de flot de données usuellement appliquées à l'étude de programmes séquentiels.

Par ailleurs, ce résultat semble susceptible de connaître des extensions. Nous avons esquissé une telle extension : l'introduction de sections critiques, une construction parallèle souvent utilisée. Nous avons étendu notre résultat de manière à englober certaines utilisations simples (mais déjà assez étendues) de sections critiques. En cette occasion, nous avons été conduit à généraliser (c'est-à-dire à affaiblir) notre exigence d'équivalence sémantique, en ouvrant la possibilité que certains calculs intermédiaires ne soient pas équivalents dans une exécution parallèle et dans la version séquentielle.

Annexe A

Démonstration du lemme 1

Le lemme 1, introduit au §4.2, précise dans quels cas, et dans quel sens, l'exécution d'une instance d'instruction α dans une exécution parallèle, dépend strictement de l'exécution d'autres instances d'instructions β telles que $\mathrm{Pre}^0(\beta, \alpha)$. (Rappelons que $\psi(\alpha)$ désigne la condition pour que α soit exécutée ou bloquée en attente ou bloquée en suspens.)

Lemme 1 Considérant un programme parallèle, pour toute exécution de ce programme, et pour toute instance α de toute instruction autre que la première instruction du programme, $\psi(\alpha)$ est pleinement déterminé par l'exécution d'une ou plusieurs instance(s) d'instruction(s) β telle(s) que $\mathrm{Pre}^0(\beta, \alpha)$. Une partie au moins de ces instances β sont spécifiées indépendamment de l'exécution considérée du programme ; les autres, s'il y en a, sont spécifiées par l'exécution des premières. Si l'une au moins de ces instances β est en erreur d'exécution, cela entraîne $\psi(\alpha) = \mathit{faux}$.

Nous avons $\mathrm{Exe}(\alpha) = \psi(\alpha)$ sauf éventuellement dans les trois cas suivants :

- α est une instance d'un WAIT w : alors, $\psi(\alpha)$ exprime la condition pour que α soit *atteinte* (ou la condition pour que α soit *atteinte ou bloquée en suspens*, dans le cas où w est à la fois un WAIT et la première instruction dans le corps d'un PDO ou dans une SECTION d'un PSECTIONS). Sous cette condition, cependant, α peut être bloquée en attente (ou bloquée en suspens), au lieu de s'exécuter, dans une situation de blocage (ou de boucle infinie dans le deuxième cas).

- α est une instance de la première instruction dans le corps d'un PDO, ou dans une SECTION d'un PSECTIONS, sans être une instance d'un WAIT : alors, $\psi(\alpha)$ exprime la condition pour α d'être exécutée ou bloquée en suspens ; cette dernière possibilité peut survenir en cas de blocage ou de boucle infinie.

- α est une instance d'un ENDWHILE : alors, $\psi(\alpha)$ exprime la condition pour α d'être exécutée ou bloquée en suspens ; cette dernière possibilité

survient lorsque le WHILE boucle indéfiniment, ou en cas de blocage ou de boucle infinie dans une itération de ce WHILE.

Preuve : Nous allons nous référer au modèle d'exécution précédemment décrit (§2.2). Nous allons examiner successivement tous les cas pouvant se présenter dans notre langage.

Nous considérons donc une instance α, d'une instruction a autre que la première du programme.

- a est un WAIT : alors, dans tous les cas intéressants, il n'est pas vrai que l'exécution de α dépend d'instances en précédence Pre^0 par rapport à α. Mais la condition pour que α soit *atteinte* – sans qu'elle soit nécessairement exécutée – se conforme à tout ce que nous allons établir maintenant, comme cela est mis en évidence en insérant fictivement une instruction CONTINUE juste avant le WAIT, et en regardant dans lequel des cas suivants se trouve ce CONTINUE. Dans ces cas suivants, on suppose donc que a n'est pas un WAIT.

- a est la première instruction dans le corps d'un PDO : alors, soit c la tête de la boucle ; soit $\mathbf{j}$ le vecteur d'indices (éventuellement vide) de c et $\mathbf{j}::k$ le vecteur d'indices de a. Pour qu'une instance $\alpha = a(\mathbf{j}::k)$ soit exécutée, il est nécessaire que l'instance correspondante $c(\mathbf{j})$ le soit, sans erreur ; inversement, l'exécution de $c(\mathbf{j})$, par l'évaluation des bornes de la boucle, caractérise entièrement quelles instances $a(\mathbf{j}::k)$ sont *atteintes ou bloquées en suspens.* Ainsi, $\psi(a(\mathbf{j}::k))$ est entièrement caractérisé par l'exécution de $c(\mathbf{j})$; et nous avons $\mathrm{Pre}^0(c(\mathbf{j}), a(\mathbf{j}::k))$. *Cependant*, une instance $a(\mathbf{j}::k)$ telle que $\psi(a(\mathbf{j}::k))$ peut être bloquée en suspens au lieu de s'exécuter, dans des situations de blocage ou de boucle infinie.

- a est la première instruction dans le corps d'un SECTION, au sein d'un PSECTIONS : alors, soit c la tête du PSECTIONS ; soit $\mathbf{j}$ le vecteur d'indices (éventuellement vide) de a et c. Pour qu'une instance $\alpha = a(\mathbf{j})$ soit *atteinte ou bloquée en suspens*, il est nécessaire et suffisant que $c(\mathbf{j})$ soit exécuté sans erreur. $\psi(\alpha)$ dépend donc entièrement d'une autre instance en précédence Pre^0 avec elle. Cependant, là encore, α peut être bloquée en suspens, dans des situations de blocage ou de boucle infinie.

- a est un ENDWHILE : alors, soit c la tête de WHILE correspondante ; soit $\mathbf{j}$ le vecteur d'indices (éventuellement vide) de a et c. Pour qu'une instance $\alpha = a(\mathbf{j})$ soit *atteinte ou bloquée en suspens*, il est nécessaire et suffisant que $c(\mathbf{j})$ soit exécuté sans erreur. $\psi(\alpha)$ dépend donc entièrement d'une autre instance en précédence Pre^0 avec elle. Cependant,

là encore, α peut être bloquée en suspens lorsque, ou bien le WHILE boucle indéfiniment, ou bien il y a une situation de blocage ou de boucle infinie dans une itération de ce WHILE.

A ce point, nous venons d'examiner les quatre cas où l'exécution d'une instance d'instruction n'est pas complètement déterminée par l'exécution d'instances d'instructions qui sont en précédence Pre^0 avec elle. Cependant, ce qui est ainsi complètement déterminé, c'est la condition pour qu'un WAIT soit *atteint* – sans que cela assure qu'il sera exécuté –; pour qu'une instance initiale d'une unité de travail soit *exécutée ou bloquée en suspens* – sans que cela assure qu'elle sera exécutée –; pour qu'une instance initiale d'une unité de travail qui se trouve être aussi un WAIT soit *bloquée en suspens ou atteinte* – sans que cela assure qu'elle sera atteinte, ou exécutée; et pour qu'un ENDWHILE soit *exécuté ou bloqué en suspens*.

Examinons maintenant les autres cas.

- a est un ENDPSECTIONS. Alors, soit c la tête de ce même PSECTIONS, et $\mathbf{j}$ le vecteur d'indices commun à c et a. L'exécution de $a(\mathbf{j})$ est entièrement déterminée par l'exécution sans erreur des instances correspondantes des instructions qui terminent toutes les sections de ce PSECTIONS. Toutes ces instances sont en précédence Pre^0 par rapport à $a(\mathbf{j})$. Elles sont spécifiées indépendamment de l'exécution particulière du programme.

- a est un ENDPDO. Alors, soit c la tête de ce même PDO et $\mathbf{j}$ le vecteur d'indices commun à c et a. Alors, la condition pour que $a(\mathbf{j})$ soit exécutée est que $c(\mathbf{j})$ soit exécutée sans erreur et que, dans le cas où l'intervalle d'itération n'est pas vide (une circonstance déterminée par l'exécution de $c(\mathbf{j})$), les instances qui terminent toutes les unités de travail parallèles correspondantes soient exécutées sans erreur. Toutes ces instances sont en précédence Pre^0 avec $a(\mathbf{j})$, et sont spécifiées par l'exécution de $c(\mathbf{j})$.

- a est la première instruction dans le corps d'un DO. Soit c la tête de ce DO, $\mathbf{j}$ le vecteur d'indices de c et $\mathbf{j}::k$ le vecteur d'indices de a. Alors, l'exécution sans erreur de $c(\mathbf{j})$ détermine complètement l'intervalle des valeurs de k qui seront considérées. Pour chacune de ces valeurs de k, la condition pour que $a(\mathbf{j}::k)$ soit exécutée est l'exécution sans erreur de $c(\mathbf{j})$ et (pour les itérations autres que la première) de l'instance de la dernière instruction du corps de la boucle, correspondant à l'itération précédente. Ces deux instances sont en précédence Pre^0 avec $a(\mathbf{j}::k)$; la première est spécifiée indépendamment de l'exécution particulière du programme, et spécifie si la seconde intervient.

- a est un ENDDO. Soit c la tête de ce DO, $\mathbf{j}$ le vecteur d'indices commun à c et a. Alors, la condition pour que $a(\mathbf{j})$ soit exécutée est que $c(\mathbf{j})$

soit exécutée sans erreur et que, dans le cas où l'intervalle d'itération n'est pas vide (une circonstance déterminée par l'exécution de $c(\mathbf{j})$), la dernière instance de la dernière itération de la boucle (spécifiée par l'exécution de $c(\mathbf{j})$) soit exécutée sans erreur. Ces deux instances sont en précédence Pre^0 avec $a(\mathbf{j})$.

- a est la première instruction de la branche THEN ou ELSE d'un IF. Soit c ce IF. L'exécution d'une instance de a est complètement déterminée par l'exécution sans erreur de l'instance correspondante de c.

- a est un ENDIF. Soit c le IF correspondant, et $\mathbf{j}$ le vecteur d'indices commun à a et c. La condition pour que $a(\mathbf{j})$ soit exécutée est l'exécution sans erreur de $c(\mathbf{j})$ et celle d'une instance – terminant la branche THEN ou la branche ELSE – spécifiée par l'exécution de $c(\mathbf{j})$. Ces deux instances sont en précédence Pre^0 avec $a(\mathbf{j})$.

- a est la première instruction dans le corps d'un WHILE, c'est-à-dire le test de la condition booléenne. Soit $\mathbf{j}$ le vecteur d'indices de la tête c du WHILE et $\mathbf{j}::k$ le vecteur d'indices de a. La condition pour que $a(\mathbf{j}::k)$ soit exécutée est l'exécution de $c(\mathbf{j})$ et (pour les itérations autres que la première) l'exécution sans erreur de l'instance, de la dernière instruction du corps du WHILE, correspondant à l'itération précédente. Ces deux instances sont en précédence Pre^0 avec $a(\mathbf{j}::k)$ et sont spécifiées indépendamment de l'exécution particulière du programme.

- a est la deuxième instruction du corps d'un WHILE, c'est-à-dire l'instruction qui suit le test que nous venons de considérer, ici noté b. Soit $\mathbf{i}$ le vecteur d'indices de b et a. L'exécution de $a(\mathbf{i})$ est complètement déterminée par l'exécution sans erreur de $b(\mathbf{i})$ (et par le résultat du test effectué par $b(\mathbf{i})$).

- Le cas restant est le plus simple : a a un prédécesseur immédiat b, de même vecteur d'indices $\mathbf{j}$, et la condition d'exécution de a est exactement l'exécution sans erreur de b.

◀

Annexe B

Démonstration des lemmes 2 et 3

Démonstration du lemme 2

Le lemme 2, qui va être démontré ici, intervient dans la démonstration du théorème 1 (chapitre 5), dans le point 1. Il suppose les hypothèses du théorème 1.

Lemme 2 Nous supposons l'hypothèse d'équivalence sémantique jusqu'à la date $\tau - 1$, et une instance d'instruction γ exécutée à τ dans $\mathbf{P}$. Pour toute instance d'instruction α telle que l'on ait $\mathrm{Exe}^{\mathsf{s}}(\alpha)$ et $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$, nous avons $\mathrm{Exe}(\alpha)$ et $\mathrm{Pre}(\alpha, \gamma)$.

Ce résultat s'applique également si, au lieu de γ, on considère une instance d'instruction exécutée avant τ dans $\mathbf{P}$.

Preuve : Nous allons démontrer le résultat concernant γ ; la dernière extension sera aisée. Nous allons faire une remarque préliminaire, puis prouverons le résultat restreint au cas où l'on a $\mathrm{Pre}^0(\alpha, \gamma)$; ensuite, nous démontrerons la généralisation au cas où l'on a $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$.

Remarque préliminaire. Dans une situation de blocage ou de boucle infinie, soit α une instance d'instruction bloquée en attente ou bloquée en suspens. Aucune instance β vérifiant $\mathrm{Pre}^0(\alpha, \beta)$ ne peut être exécutée. Cela résulte aisément du modèle d'exécution et de la définition de Pre^0. (En d'autres termes, l'exécution ne peut pas *contourner* un blocage ou une boucle infinie.)

Considérons Pre^0. Nous avons donc $\mathrm{Exe}^{\mathsf{s}}(\alpha)$ et $\mathrm{Pre}^0(\alpha, \gamma)$. Nous avons clairement $\mathrm{Pre}(\alpha, \gamma)$, car la précédence de contrôle Pre^0 est commune à toutes les exécutions (§3.3.1). Supposons qu'une instance d'instruction α exécutée dans $\mathbf{S}$ et vérifiant $\mathrm{Pre}^0(\alpha, \gamma)$, ne s'exécute pas dans $\mathbf{P}$ (alors, α n'est pas la première instruction du programme : le lemme 1 s'applique à α). Nous allons démontrer que, dans ce cas, une autre instance d'instruction α_1 vérifiant

$\mathrm{Pre}^0(\alpha_1, \alpha)$ est, également, exécutée dans **S** et non dans **P**, ce qui conduira ensuite à une contradiction.

Appliquons le lemme 1 à l'exécution de α dans **S**. Selon les hypothèses *(ii)* et *(iii)* du théorème 1, il n'y a pas d'instance d'instruction *bloquée en attente* ni *bloquée en suspens* dans **S**. Donc selon le lemme 1, $\mathrm{Exe}^{\mathsf{s}}(\alpha)$ dépend d'une ou plusieurs instance(s) β_i exécutée(s) dans **S**, vérifiant $\mathrm{Pre}^0(\beta_i, \alpha)$, et donc $\mathrm{Pre}^0(\beta_i, \gamma)$. Par conséquent, si tous ces β_i étaient exécutés dans **P**, ils le seraient avant la date τ, d'où l'équivalence sémantique, qui impliquerait que α serait atteint ou en suspens dans **P**. Par conséquent, dans **P**, la non exécution de α impliquerait l'une de deux conséquences. Ou bien tous les β_i sont effectivement exécutés dans **P** mais α est bloqué en attente ou en suspens, participant ainsi à une situation de blocage ou de boucle infinie. Cette possibilité est exclue par la remarque préliminaire ci-dessus : le fait d'avoir $\mathrm{Pre}^0(\alpha, \gamma)$ empêcherait l'exécution de γ, qui est pourtant supposée. Il reste la possibilité qu'au moins l'un des β_i ne s'exécute pas dans **P** : notons-le α_1.

Ainsi, la non exécution de α dans **P** impliquerait qu'une autre instance, α_1, en précédence Pre^0 avec α, ne serait pas exécutée non plus dans **P**, bien qu'elle le soit dans **S**.

Cet argument peut être répété pour α_1 : ainsi, nous aurions une suite infinie ($\alpha_0 = \alpha$, α_1, α_2,...) telle que chaque α_i serait exécuté dans **S** et précédé, selon Pre^0, par α_{i+1}. Cela contredit l'observation simple selon laquelle il y a un nombre fini de dates d'exécution entre le démarrage du programme et tout point qu'il atteint, dans toute exécution [1].

L'extension à $\mathrm{Pre}^{\mathsf{s}}$. Supposons qu'une instance α vérifiant $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$ et non $\mathrm{Pre}^0(\alpha, \gamma)$ s'exécute dans **S**. $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$ est réalisé à travers des synchronisations, c'est-à-dire, comme nous l'avons vu (§3.3.2), par un ou plusieurs chemin(s) de la forme :

$$\alpha \to \pi_1 \text{ ou } \alpha = \pi_1$$

$$\pi_1 \rightsquigarrow \omega_1 \to \pi_2 \rightsquigarrow \omega_2 \to \cdots \to \pi_n \rightsquigarrow \omega_n$$

$$\omega_n \to \gamma$$

où, de nouveau, $\to$ désigne une relation Pre^0, π_i désigne un POST, ω_i désigne un WAIT et $\rightsquigarrow$ désigne une relation de synchronisation $\mathrm{Sync}^{\mathsf{s}}$; en outre, tous les π_i et ω_i sont exécutés dans **S** (se souvenir de la "fermeture transitive modulo $\mathrm{Exe}^{\mathsf{s}}$" impliquée dans $\mathrm{Pre}^{\mathsf{s}}$). Nous avons $\omega_n \to \gamma$ et non $\omega_n = \gamma$ en raison de la restriction que nous avons introduite (§3.3.2) dans la définition de $\mathrm{Pre}^{\mathsf{s}}$.

1. Dans ce raisonnement, il est essentiel d'avoir $\mathrm{Exe}^{\mathsf{s}}(\alpha_i)$, en même temps que $\mathrm{Pre}^0(\alpha_{i+1}, \alpha_i)$, pour obtenir la contradiction, car l'ordre Pre^0 n'est pas *bien fondé* (en raison de la manière dont les boucles DO et PDO génèrent des ensembles d'instances s'étendant à l'infini *dans les deux sens*).

Nous avons $\omega_n \rightarrow \gamma$, c'est-à-dire $\mathrm{Pre}^0(\omega_n, \gamma)$; par conséquent, selon la première partie de ce lemme, ω_n s'exécute avant la date τ dans **P**. Par récurrence en remontant, nous allons montrer que tous les π_i et ω_i, et finalement α, s'exécutent avant τ dans **P**. Supposons que ω_i s'exécute avant τ. Alors, l'hypothèse d'équivalence sémantique jusqu'à $\tau - 1$ s'applique à ω_i et à toute variable présente dans ω_i, donc à l'événement présent dans ω_i, ε_{ω_i} : tous les calculs de ε_{ω_i} effectués avant l'exécution de ω_i sont identiques dans les deux exécutions, et s'y sont effectués dans le même ordre. Par conséquent, comme nous avons $\mathrm{Sync}^{\mathsf{s}}(\pi_i, \omega_i)$, π_i s'est exécuté dans **P** avant ω_i et c'est son exécution qui a rendu possible celle de ω_i, dans **P** comme dans **S** : nous avons $\mathrm{Sync}(\pi_i, \omega_i)$ (§3.3.3).

Considérons maintenant le cas $i > 1$ et déduisons l'exécution de ω_{i-1}. Etant donné que nous avons $\mathrm{Pre}^0(\omega_{i-1}, \pi_i)$ et $\mathrm{Exe}^{\mathsf{s}}(\omega_{i-1})$, selon la première partie du lemme, ω_{i-1} s'exécute avant π_i, par conséquent avant la date τ, dans **P**. Nous concluons donc que ω_1 s'exécute avant τ dans **P**. Le raisonnement ci-dessus assure que π_1 aussi s'exécute avant τ dans **P**. Maintenant, nous avons ou bien $\mathrm{Pre}^0(\alpha, \pi_1)$ (et $\mathrm{Exe}^{\mathsf{s}}(\alpha)$), ou bien $\alpha = \pi_1$, ce qui implique que α s'exécute effectivement avant τ dans **P** ; en outre, nous avons $\mathrm{Pre}(\alpha, \gamma)$, par fermeture transitive modulo Exe.

◀

Démonstration du lemme 3

Le lemme 3, qui va être démontré ici, intervient dans la démonstration du théorème 1 (chapitre 5), dans le point 5. Il suppose les hypothèses du théorème 1.

Lemme 3 Nous supposons la propriété d'équivalence sémantique le long de **P**. Soit une instance de WAIT γ bloquée en attente dans **P**. Pour toute instance d'instruction α telle que l'on ait $\mathrm{Exe}^{\mathsf{s}}(\alpha)$ et $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$, nous avons $\mathrm{Exe}(\alpha)$ et $\mathrm{Pre}(\alpha, \gamma)$.

Preuve : La démonstration est très proche de celle du lemme 2 ci-dessus. Nous complétons d'abord la remarque préliminaire qui ouvre la preuve du lemme 2. Nous avions observé ceci : α étant une instance d'instruction bloquée en attente ou bloquée en suspens, aucune instance β vérifiant $\mathrm{Pre}^0(\alpha, \beta)$ ne peut être exécutée. Une telle instance β ne peut pas davantage être un WAIT atteint et bloqué en attente. Cela résulte du modèle d'exécution et de la définition de Pre^0.

Moyennant cette remarque, la preuve du lemme 2 se transpose très simplement ici.

◀

Annexe C

Communication à la conférence PARLE 94

Parallel Architectures and Languages Europe, 1994. [5]

Pour des raisons de copyright, cette communication n'est pas accessible ici. Elle figure dans la version papier de cette thèse.

Annexe D

Lexique français-anglais

(ordre) bien fondé	well-founded (order)
blocage	deadlock
bloqué en suspens	persistently pending
conflit mémoire	data race
conservatrice (approximation)	conservative (approx.)
correction	correctness
détermination	determinacy
emboîté	nested
en attente	waiting
en suspens	pending
entrée	input
erreur d'exécution	execution fault
événement (type)	event
instance	instance
instruction	statement
ordre d'indirection	indirection order
processeur	processor
processus	process
sortie	output
tête	header
unité de travail	unit of work
verrou	lock

Bibliographie

[1] A. Aho, R. Sethi, and J. D. Ullman. *Compilers*. Addison-Wesley, 1986.

[2] L. Bougé. Sémantiques du parallélisme : un tour d'horizon. *http://www.ens-lyon.fr/~bouge/Research*, Juillet 1988.

[3] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 21–30, Seattle, march 1990. ACM Press.

[4] G. Caplain, R. Lalement, and T. Salset. Semantic analysis of a control-parallel extension of Fortran. Technical Report 93-18, CERMICS, 1993.

[5] G. Caplain, R. Lalement, and T. Salset. Checking the serial correctness of control-parallel programs. In *Parallel Architectures and Languages Europe*, pages 741–744, Athens, Greece, July 1994. Springer Verlag, LNCS 817.

[6] F. Coelho. *Contributions à la compilation du HPF*. Thèse, Ecole des Mines de Paris, octobre 1996.

[7] J.F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, July 1995.

[8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[9] B. Creusillet. *Analyses de Régions de tableaux et Applications*. Thèse, Ecole des Mines de Paris, décembre 1996.

[10] B. Creusillet and F. Irigoin. Interprocedural analyses of fortran programs. Technical Report A–303, Ecole des Mines de Paris, juin 1997.

[11] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[12] M.B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs.* PhD thesis, University of Massachusetts, Amherst, MA, September 1995.

[13] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journ. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[14] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[16] A. Geist, A. Beguelin, J. Dongarra, Weicheng Jiang, R. Manchek, and V. Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1994.

[17] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[18] M.C. Giboulot and F. Thomasset. Automatic parallelization of structured if statements without if conversion. Technical Report RR–1408, INRIA, juin 1991.

[19] C.A. Gunter. *Semantics of Programming Languages.* MIT Press, 1992.

[20] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[21] D. Hirschkoff. Bisimulation proofs for the pi-calculus in the calculus of constructions. Technical Report 96-62, CERMICS, 1996.

[22] D. Hirschkoff. Up-to context proofs for the pi-calculus in the Coq system. Technical Report 96-82, CERMICS, 1996.

[23] C. A. R. Hoare. Communicating sequential processes. *Communications of the Association for Computing Machinery*, 21(8):666–677, August 1978.

[24] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance all-software distributed shared memory. In *ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 213–228. ACM Press, December 1995.

[25] R. Lalement. *Logique, réduction, résolution.* Masson, 1990.

[26] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. ACM*, 21:558–564, 1978.

[27] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C–28:690–691, 1979.

[28] S.P. Masticola. *Static detection of deadlocks in polynomial time.* PhD thesis, Rutgers University, May 1993.

[29] R. Milner. The polyadic π-calculus: a tutorial. Technical report, LFCS, Computer Science Department, University of Edinburgh, October 1991.

[30] S. Muchnick. *Advanced Compiler Design Implementation.* Morgan & Kaufman, 1998.

[31] S. Muchnick and N. Jones. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[32] OpenMP: A proposed industry standard API for shared memory programming. Technical report, October 1997.

[33] C. Pancake. *Parallel Processing Model for High Level Programming Languages.* ANSI, March 1992. (Proposed Standard).

[34] C.H. Papadimitriou. The serializability of concurrent database updates. *J.ACM*, 26:631–653, 1979.

[35] M. Raynal. About logical clocks for distributed systems. *Publication interne IRISA n^o 607*, octobre 1991.

[36] Thierry Salset. *Correction séquentielle de programmes parallèles dans le modèle asynchrone et mémoire partagée.* Thèse, Ecole Nationale des Ponts et Chaussées, juillet 1997.

[37] J. Subhlok. *Analysis of Synchronization in a Parallel Programming Environment.* PhD thesis, Rice University, April 1998.

[38] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[39] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Mass., 1993.

[40] H. Zima with B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.

[41] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Mass., 1989.

[42] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[43] X3H5. *FORTRAN 77 Binding of X3H5 Model for Parallel Programming Constructs*. ANSI, September 1992. (draft version).